Valence

unknown

Mar 17, 2025

CONCEPTS

Valence is a tool designed to make your life easier when it comes to building data integrations with Salesforce. We're happy to get you started with it.

The first thing you should understand is that Valence installs into a single Salesforce org and does all of its magic from inside that org. There is no external middleware server involved, which means data moves directly between your Salesforce instance and whatever system you want to exchange information with.

We've designed Valence to be intuitive for admins to use, and easy to extend when custom functionality is needed.

CHAPTER

ONE

CONCEPTS

Start here to develop a basic understanding of the component parts of the Valence engine.

- *Overview* — Learn about the basic building blocks of Valence and how data movement happens.
- *Metadata Files* — Learn how Valence configurations can be moved between Salesforce environments.
- *External Systems* — Read about what you can connect to and how the connection is made.
- *Delta Sync* — Understand how Valence fetches records based on time since the last fetch.
- *Testing Mode* — Learn about doing test runs that don't change your target system.
- *Handling Errors* — How does Valence help you when things go awry?
- *Invoking Valence* — What are ways to start Link runs other than scheduling?
- *Release Notes* — Valence release notes.

CHAPTER

TWO

ADVANCED CONCEPTS

Once you're comfortable with the basics, dig into some wonky stuff with our advanced topics.

- *Schema* — Understand how Valence discovers record shapes and ensures schema accuracy.
- *Multidimensional Data* — Tackle more complex data shapes that have nested objects and arrays.
- *Link Splits* — Start delivering records to more than one target.

CHAPTER

THREE

EXTENSION DEVELOPMENT

Thinking about writing your own Adapter or Filter extension to add custom functionality to Valence?

- *Extensions* — Learn the basics of how Valence Extensions are defined and assembled.
- *Property Path* — Reading from and writing to records.
- *Configurability* — Get comfortable with how to make your Extension user-configurable within the Valence UI.
- *Configuration Structure* — Using a specialized structure to let Valence build configuration forms for you.
- *Configuration Component* — Using your own Lightning component to let Users configure your extension.
- *Writing Test Coverage* — Get instructions and see examples for writing Apex test coverage for Extensions.

CHAPTER

FOUR

GO DEEPER ON EACH EXTENSION TYPE

- *Creating An Adapter* — Step by step explanation for how to build a high quality Valence Adapter.
- *Adapter Authentication* — Figure out your options for authenticating with external systems.
- *Additional Table Details* — Store additional schema details about a table for runtime retrieval.
- *Creating A Filter* — Step by step explanation for how to build a high quality Valence Filter.

CHAPTER FIVE

GUIDES

We have guides to help you understand certain topics in more detail:

- *Sending Realtime Records To Salesforce*
- *Evaluating an API for Ease-of-Use With Valence*
- *How to Design a Complementary Valence API*
- *Using Valence Between Two Salesforce Orgs (aka Salesforce to Salesforce)*
- *How To Stop Infinite Loops In Bidirectional Syncs*
- *Building a Configurable Filter that Ignores Records Based on User-selected Cutoff Date*

5.1 Overview

Valence is a middleware engine that lives inside your Salesforce org and moves data between your org and other *external systems and services*.

You configure it inside Salesforce's user interface (works in both Classic and Lightning), and moving the configuration *Metadata Files* between different Salesforce environments is simple and straightforward. That means you can build and test Valence in a sandbox, and easily promote the entire configuration (or parts of it) to your production org. You can also export these configuration files from your org to keep backups.

Valence was built to *handle large data volumes*; people move millions of rows of data in and out of Salesforce with it.

Integrations are a complex space, so Valence contains many layers of error handling and diagnostics to help you navigate what is happening with your data movement.

Every integration has its own little wrinkles and intricacies, so we've focused on a balance between features and capacities that every integration wants, and hooks for extensibility throughout the engine when special adaptation is required. Regardless of what extensions you add to your setup, Valence is designed for admins to always be able to declaratively design what data moves, where, when, and how. The way extensions hook into the engine allows them to be configured and used within the normal Valence user interface screens.

Valence is the distillation of many years of experience working with integrations on the Salesforce platform, and is tuned to understand the nuances of the platform better than any other integration product. That means you can do things like deliver parent and child records out of order, populate Master-Detail fields based on external identifiers, and upsert on formula fields.

Let's dive in and break down how record movement works using Valence.

5.1.1 How Does Data Move?

The basic building block in Valence is something called a **Link**, which you can think of as a pipe that sits between Salesforce and an external system. Data flows into one end of the pipe and out the other end.

More specifically, a Link represents a connection between an object in Salesforce and a table/object somewhere else. For example, you might have the Company table in your ERP connected to the Account object in Salesforce. That would be a single Link. If you connected the Person table from this fictitious ERP to the Contact object in Salesforce that would be a second Link.

A Link is a container for many additional components that are working together to move data from A to B, and has its own mappings, schedule, settings, configurations and analytics.

If we were to look inside a Link "pipe" you'd see that it looks something like this:

1. Source Adapter
2. Filter A
3. Filter B
4. Filter C
5. Filter . . .
6. Target Adapter

Adapters

An **Adapter** knows how to work with a system or data store. Analogy: an Adapter is like a translator; if an Adapter can speak Latin then Valence can work with that Adapter to change English to Latin and vice-versa.

Every Link defines two Adapters that it will use: a **source** Adapter to get data from, and a **target** Adapter to send data to. Some Adapters can only be used as a source (read from a system but not write to it), some only as a target (write to a system but not read from it), and some can be either.

Example Adapter Use Cases

- An Adapter that can talk to Quickbooks (or SAP, or Netsuite, or any number of business applications and services)
- An Adapter that can talk to back-office databases your IT team manages
- An Adapter that can talk to a second Salesforce org
- An Adapter that can speak GraphQL
- An Adapter that can speak REST or SOAP

Filters

A **Filter** processes a data record as it moves through the Valence engine. It is an opportunity to observe and possibly manipulate records as they go by. Any number of Filters can be attached to a Link, and the order in which they fire (and any configuration of them) is controlled by the admin setting up the Link.

Example Filter Use Cases

- Adding a constant value to records going by
- Manipulating date/time strings to be friendly to the target system
- Filtering out records that should be ignored and not processed
- Validating records for custom business logic that would add a warning or an error to a record
- Transforming record shape or field values

Getting More Adapters and Filters

Valence comes with certain Adapters and Filters out of the box. You can get more from:

1. Our open-source Adapters on GitHub
2. Our open-source Filters on GitHub
3. Read our documentation on *Creating An Adapter* or *Creating A Filter*
4. Custom-built for you by Salesforce implementation partners with Valence expertise
5. Prebuilt from businesses that write Valence extensions as a product offering

We're happy to answer questions about what Adapters and Filters exist and where to find them if you write to our team at success@valence.app.

5.1.2 When Does Data Move?

At this point we understand that a Link is a like a pipe that moves record data from one system to another, and we have a sense of what elements participate in that movement. But *when* do records move?

Valence supports both scheduled and real-time data movement, inbound to Salesforce or outbound from Salesforce. Whether a Link is scheduled, real-time, or on-demand will depend on what trigger conditions you configure for it. It can even be a mix of all three!

There are a large variety of trigger conditions that will cause a Link "run" (some records moving from A to B).

1. The Link is on a schedule and it's time to run again
2. The Link is configured to run after another Link, and that other Link just finished a run
3. The Link was invoked by an Apex Trigger that fired
4. The Link was invoked as part of a Salesforce Process or Flow
5. Fresh data was delivered from an external system
6. An admin clicked the "Run Now" button
7. An admin uploaded a CSV file with new records
8. The Link was invoked programmatically
9. A Link Split uses this Link as a secondary delivery target and the primary Link just finished

5.1.3 What Happened During Movement?

Some of the big questions with any integration are: what happened? How did it go?

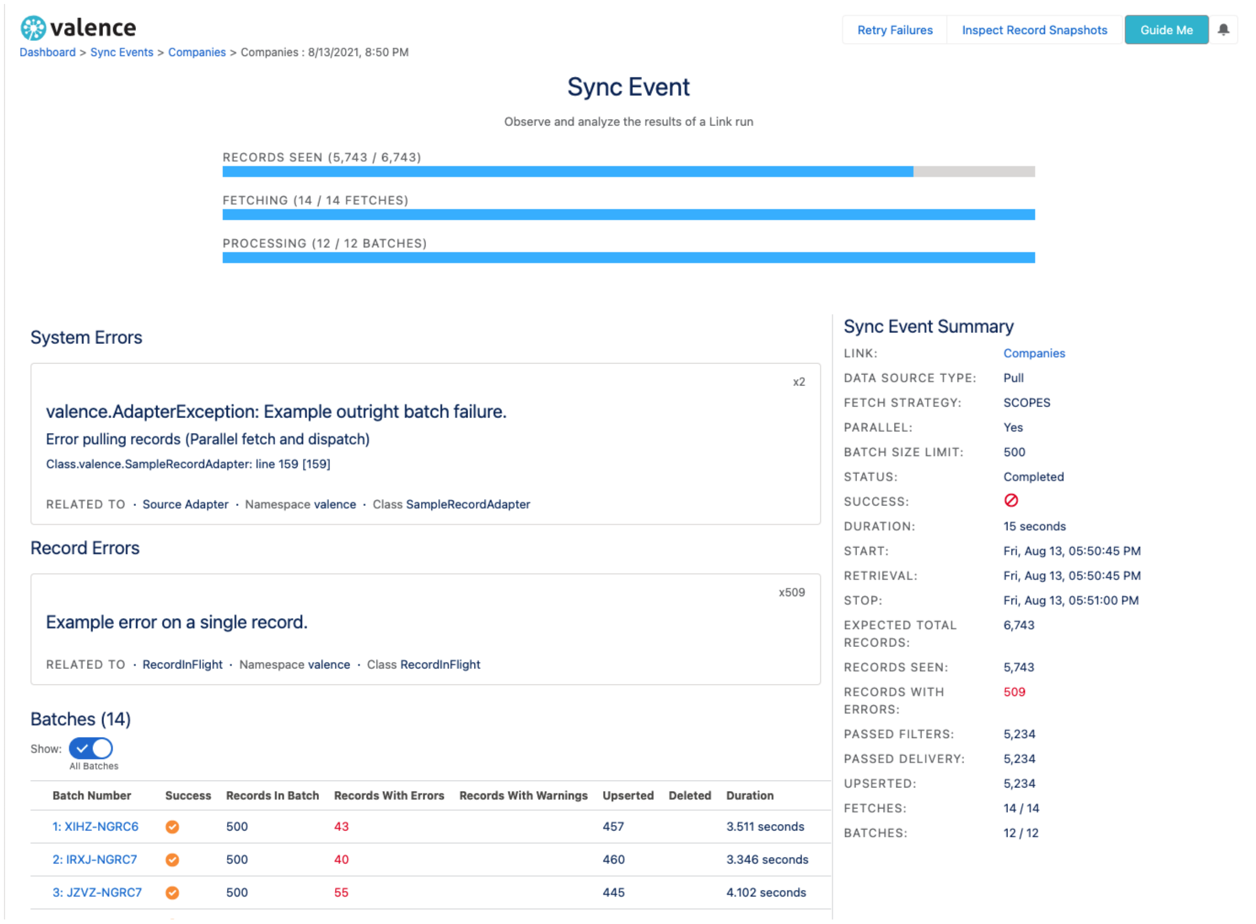
Valence is very good at answering these kinds of questions.

Whenever a Link runs, an SObject record called a **Sync Event** is generated. This record holds all kinds of information about what happened during the run, like:

- How many records did we expect to move?
- How many records did we move?
- How many records had errors? warnings?
- How many records were ignored?
- How many records were successfully delivered?
- What errors occurred and which records had those errors?
- How long did this take?
- Which fields did we see? What were their population percentages?
- And many more

Tip: Since a Sync Event is an SObject, you can use normal Salesforce features to report on it, put it in a dashboard, an email, or anything else you'd like to do in order to review movement or be notified when certain things happen.

In the Valence user interface, you can see this information by looking at the Sync Event Summary screen. This screen will give you a live, real-time picture of the run as it is happening and summary information afterwards.


valence
Retry Failures
Inspect Record Snapshots
Guide Me
Dashboard > Sync Events > Companies > Companies : 8/13/2021, 8:50 PM

Sync Event

Observe and analyze the results of a Link run

RECORDS SEEN (5,743 / 6,743)

FETCHING (14 / 14 FETCHES)

PROCESSING (12 / 12 BATCHES)

System Errors

x2

valence.AdapterException: Example outright batch failure.

Error pulling records (Parallel fetch and dispatch)

Class.valence.SampleRecordAdapter: line 159 [159]

RELATED TO · Source Adapter · Namespace valence · Class SampleRecordAdapter

Record Errors

x509

Example error on a single record.

RELATED TO · RecordInFlight · Namespace valence · Class RecordInFlight

Batches (14)

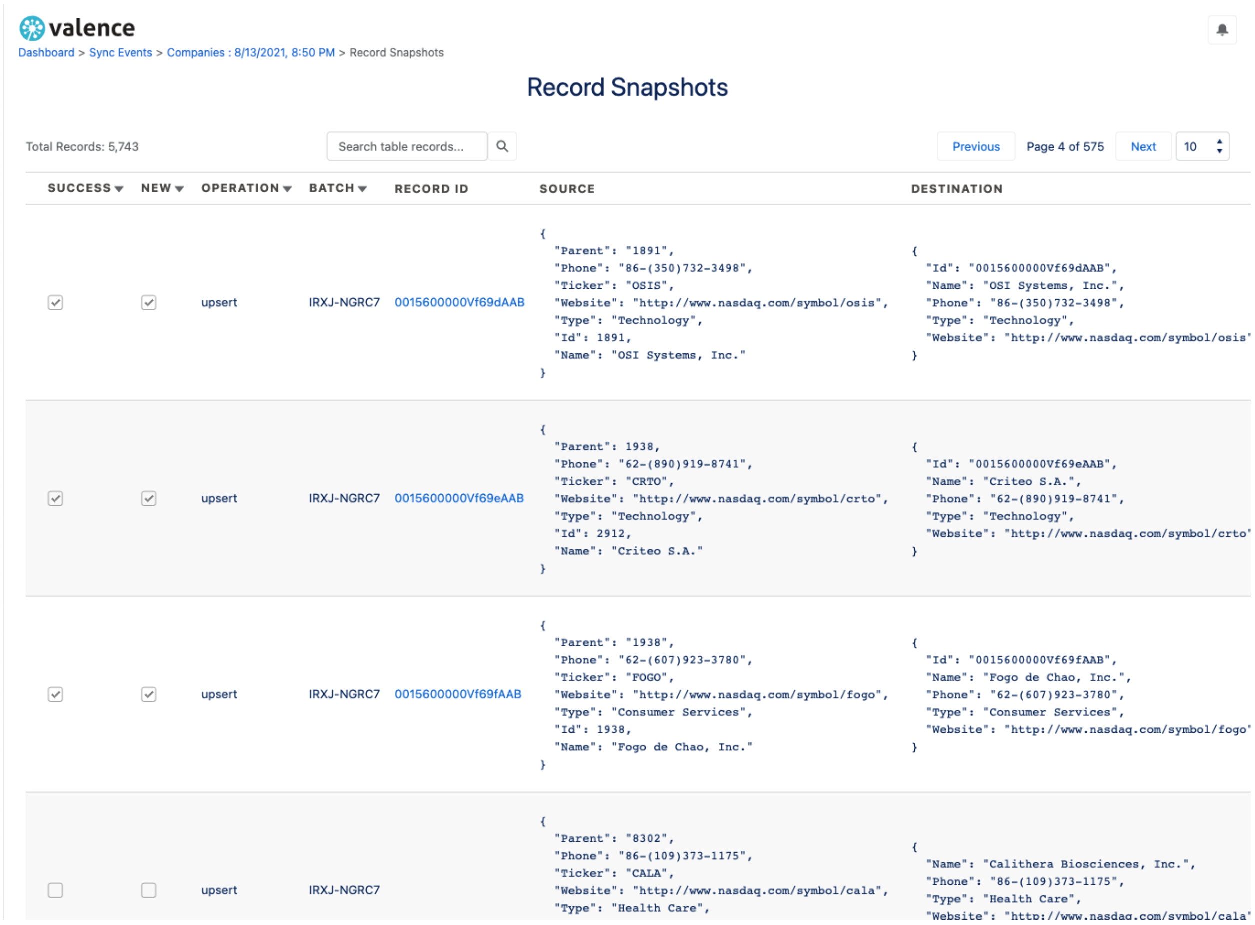
Show: All Batches

Batch Number	Success	Records In Batch	Records With Errors	Records With Warnings	Upserted	Deleted	Duration
1: XIHZ-NGRC6	✓	500	43		457		3.511 seconds
2: IRXJ-NGRC7	✓	500	40		460		3.346 seconds
3: JZVZ-NGRC7	✓	500	55		445		4.102 seconds

Sync Event Summary

LINK:	Companies
DATA SOURCE TYPE:	Pull
FETCH STRATEGY:	SCOPES
PARALLEL:	Yes
BATCH SIZE LIMIT:	500
STATUS:	Completed
SUCCESS:	⊘
DURATION:	15 seconds
START:	Fri, Aug 13, 05:50:45 PM
RETRIEVAL:	Fri, Aug 13, 05:50:45 PM
STOP:	Fri, Aug 13, 05:51:00 PM
EXPECTED TOTAL RECORDS:	6,743
RECORDS SEEN:	5,743
RECORDS WITH ERRORS:	509
PASSED FILTERS:	5,234
PASSED DELIVERY:	5,234
UPSERTED:	5,234
FETCHES:	14 / 14
BATCHES:	12 / 12

In addition to metadata about the run itself, you can also inspect individual records in their before and after states to understand how they were transformed and why you are seeing certain error messages.


valence
Dashboard > Sync Events > Companies : 8/13/2021, 8:50 PM > Record Snapshots

Record Snapshots

Total Records: 5,743 Search table records... Previous Page 4 of 575 Next 10

SUCCESS	NEW	OPERATION	BATCH	RECORD ID	SOURCE	DESTINATION
✓	✓	upsert	IRXJ-NGRC7	0015600000Vf69dAAB	{ "Parent": "1891", "Phone": "86-(350)732-3498", "Ticker": "OSIS", "Website": "http://www.nasdaq.com/symbol/osis", "Type": "Technology", "Id": 1891, "Name": "OSI Systems, Inc." }	{ "Id": "0015600000Vf69dAAB", "Name": "OSI Systems, Inc.", "Phone": "86-(350)732-3498", "Type": "Technology", "Website": "http://www.nasdaq.com/symbol/osis' }
✓	✓	upsert	IRXJ-NGRC7	0015600000Vf69eAAB	{ "Parent": 1938, "Phone": "62-(890)919-8741", "Ticker": "CRTO", "Website": "http://www.nasdaq.com/symbol/crto", "Type": "Technology", "Id": 2912, "Name": "Criteo S.A." }	{ "Id": "0015600000Vf69eAAB", "Name": "Criteo S.A.", "Phone": "62-(890)919-8741", "Type": "Technology", "Website": "http://www.nasdaq.com/symbol/crto' }
✓	✓	upsert	IRXJ-NGRC7	0015600000Vf69fAAB	{ "Parent": "1938", "Phone": "62-(607)923-3780", "Ticker": "FOGO", "Website": "http://www.nasdaq.com/symbol/fogo", "Type": "Consumer Services", "Id": 1938, "Name": "Fogo de Chao, Inc." }	{ "Id": "0015600000Vf69fAAB", "Name": "Fogo de Chao, Inc.", "Phone": "62-(607)923-3780", "Type": "Consumer Services", "Website": "http://www.nasdaq.com/symbol/fogo' }
☐	☐	upsert	IRXJ-NGRC7		{ "Parent": "8302", "Phone": "86-(109)373-1175", "Ticker": "CALA", "Website": "http://www.nasdaq.com/symbol/cala", "Type": "Health Care",	{ "Name": "Calithera Biosciences, Inc.", "Phone": "86-(109)373-1175", "Type": "Health Care", "Website": "http://www.nasdaq.com/symbol/cala'

5.1.4 Salesforce Limits (or Lack Thereof)

Anyone working with Salesforce knows that every automated process on the platform has to operate within the governor limits defined by Salesforce. These limits can be quite restrictive, so how can a platform-native middleware engine move millions of records without running into trouble?

Valence was designed and built from the ground up to be highly-parallel and operate across many different execution contexts as needed. In other words, that's like having a magic lamp and your third wish is always to ask for three more wishes. This load balancing is done internally by the engine and typically you won't have to think about it or know about it.

Here's an example: the maximum number of records that can be written to the Salesforce database in a single execution context is 10,000. So what happens if an external system hands Valence 15,000 rows in one fell swoop?

All 15,000 records will be written to the database by Valence. Valence will detect that it has too many records for a single write, and it will split the records into batches across different execution contexts so that no limits are exceeded.

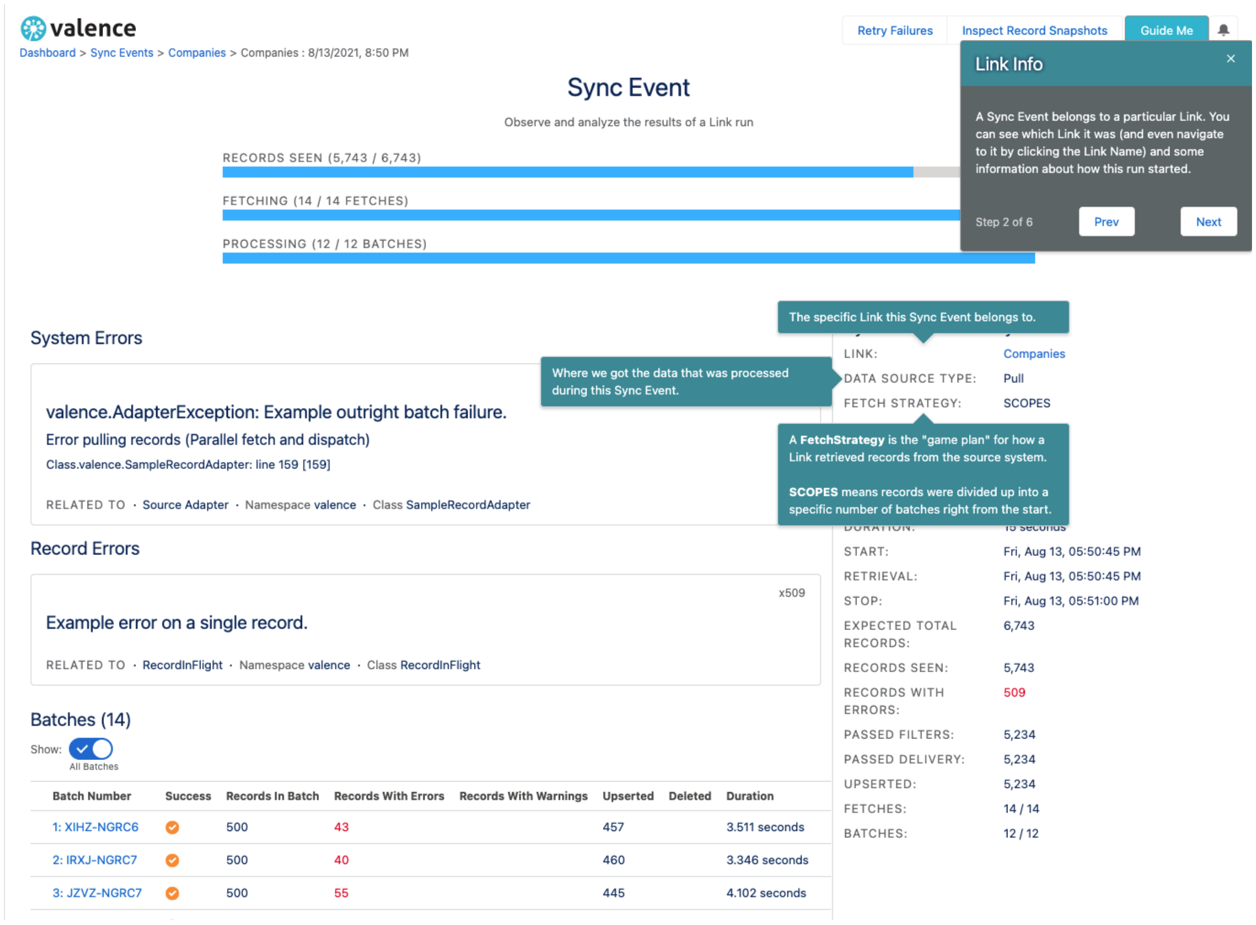
From the perspective of both the admin user and extension developers, Valence abstracts away the Salesforce governor limits so you don't have to think about them when designing your data flows with Valence.

5.1.5 Getting Help Inside the Application

There is a comprehensive built-in help system inside of the Valence user interface that will help you understand each screen and what you can do there.

Look for the `Guide Me` button on the top right.

It will open an overlay that you can flip through to see different cards and popups explaining facets of what you are seeing on the screen.



Dashboard > Sync Events > Companies > Companies : 8/13/2021, 8:50 PM

Sync Event

Observe and analyze the results of a Link run

RECORDS SEEN (5,743 / 6,743)

FETCHING (14 / 14 FETCHES)

PROCESSING (12 / 12 BATCHES)

Link Info

A Sync Event belongs to a particular Link. You can see which Link it was (and even navigate to it by clicking the Link Name) and some information about how this run started.

Step 2 of 6 Prev Next

The specific Link this Sync Event belongs to.

Where we got the data that was processed during this Sync Event.

A **FetchStrategy** is the "game plan" for how a Link retrieved records from the source system.

SCOPES means records were divided up into a specific number of batches right from the start.

System Errors

valence.AdapterException: Example outright batch failure.

Error pulling records (Parallel fetch and dispatch)

Class.valence.SampleRecordAdapter: line 159 [159]

RELATED TO · Source Adapter · Namespace valence · Class SampleRecordAdapter

Record Errors

x509

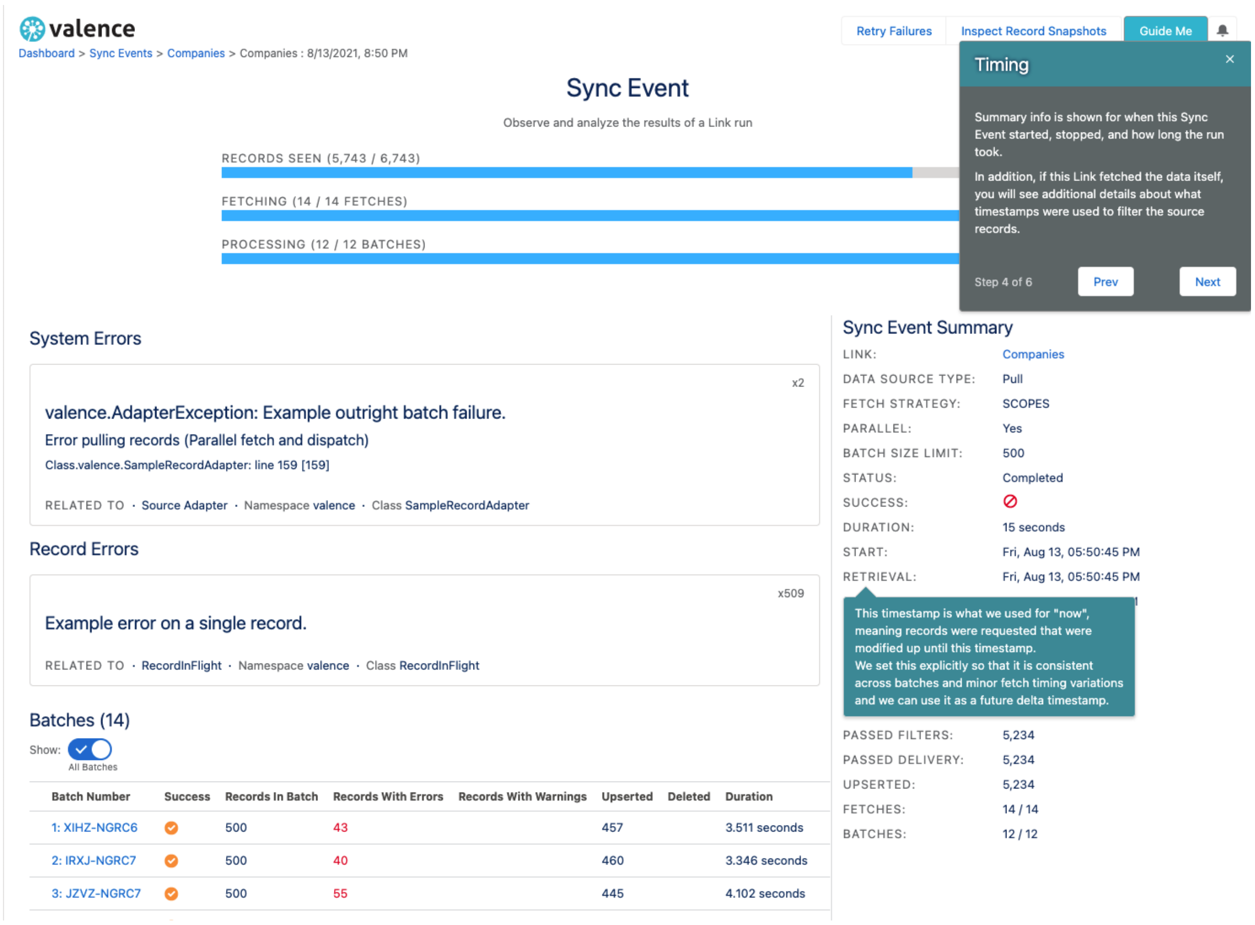
Example error on a single record.

RELATED TO · RecordInFlight · Namespace valence · Class RecordInFlight

Batches (14)

Batch Number	Success	Records In Batch	Records With Errors	Records With Warnings	Upserted	Deleted	Duration
1: XIHZ-NGRC6	✓	500	43		457		3.511 seconds
2: IRXJ-NGRC7	✓	500	40		460		3.346 seconds
3: JZVZ-NGRC7	✓	500	55		445		4.102 seconds

LINK:	Companies
DATA SOURCE TYPE:	Pull
FETCH STRATEGY:	SCOPES
START:	Fri, Aug 13, 05:50:45 PM
RETRIEVAL:	Fri, Aug 13, 05:50:45 PM
STOP:	Fri, Aug 13, 05:51:00 PM
EXPECTED TOTAL RECORDS:	6,743
RECORDS SEEN:	5,743
RECORDS WITH ERRORS:	509
PASSED FILTERS:	5,234
PASSED DELIVERY:	5,234
UPSERTED:	5,234
FETCHES:	14 / 14
BATCHES:	12 / 12



valence

Dashboard > Sync Events > Companies > Companies : 8/13/2021, 8:50 PM

Retry Failures · Inspect Record Snapshots · Guide Me

Sync Event

Observe and analyze the results of a Link run

RECORDS SEEN (5,743 / 6,743)

FETCHING (14 / 14 FETCHES)

PROCESSING (12 / 12 BATCHES)

Timing

Summary info is shown for when this Sync Event started, stopped, and how long the run took.

In addition, if this Link fetched the data itself, you will see additional details about what timestamps were used to filter the source records.

Step 4 of 6 · Prev · Next

System Errors

x2

valence.AdapterException: Example outright batch failure.

Error pulling records (Parallel fetch and dispatch)

Class.valence.SampleRecordAdapter: line 159 [159]

RELATED TO · Source Adapter · Namespace valence · Class SampleRecordAdapter

Record Errors

x509

Example error on a single record.

RELATED TO · RecordInFlight · Namespace valence · Class RecordInFlight

Batches (14)

Show: All Batches

Batch Number	Success	Records In Batch	Records With Errors	Records With Warnings	Upserted	Deleted	Duration
1: XIHZ-NGRC6	✓	500	43		457		3.511 seconds
2: IRXJ-NGRC7	✓	500	40		460		3.346 seconds
3: JZVZ-NGRC7	✓	500	55		445		4.102 seconds

Sync Event Summary

LINK:	Companies
DATA SOURCE TYPE:	Pull
FETCH STRATEGY:	SCOPES
PARALLEL:	Yes
BATCH SIZE LIMIT:	500
STATUS:	Completed
SUCCESS:	⊘
DURATION:	15 seconds
START:	Fri, Aug 13, 05:50:45 PM
RETRIEVAL:	Fri, Aug 13, 05:50:45 PM
PASSED FILTERS:	5,234
PASSED DELIVERY:	5,234
UPSERTED:	5,234
FETCHES:	14 / 14
BATCHES:	12 / 12

This timestamp is what we used for "now", meaning records were requested that were modified up until this timestamp.
We set this explicitly so that it is consistent across batches and minor fetch timing variations and we can use it as a future delta timestamp.

5.2 Metadata Files

The basic configuration building blocks of Valence (Links, Filters, Mappings, and related records) are all stored as Custom Metadata Type records. This means that they can easily be extracted from a Salesforce org, tracked in version control, or moved between environments.

If you are comfortable working with Salesforce XML metadata files, feel free to extract and work with these files directly. If you prefer a more polished experience, Valence includes a **Change Set Editor** that will allow you to build Change Sets within Valence, for upload to other environments (such as moving from sandbox to production).

Because all the Valence configuration exists as cMDT records, you can easily set up Valence in a sandbox, test it thoroughly, then move the exact same setup to production with a Change Set or your preferred deploy mechanism.

5.2.1 Valence Custom Metadata Types

API Name	Label	Description
valence__ValenceAdapter__mdt	Valence Adapter	Lists Valence Adapters that are installed in this environment and are available to be used. An adapter allows us to talk to an external system.
valence__ValenceConfiguration__mdt	Valence Configuration	Global configuration settings for Valence.
valence__ValenceDataLink__mdt	Valence Data Link	Encapsulates the idea of two tables in different systems being linked to each other, and having records move between those tables.
valence__ValenceDataLinkFilter__mdt	Valence Data Link Filter	Junction table that connects Links to Filters.
valence__ValenceDataLinkSplit__mdt	Valence Data Link Split	Connects a Link that receives data with an additional Link it will send that data to. Allows a single Link run to deliver to multiple places.
valence__ValenceField__mdt	Valence Field	Represents a field that is part of the source or target schema for a Link.
valence__ValenceFilter__mdt	Valence Filter	Filters are applied to records in flight to transform them or take special action on them. This table lists available filters to be used.
valence__ValenceFilterConfig__mdt	Valence Filter Config	Child record of a LinkFilter allowing configurations of that Filter's behavior when it runs.
valence__ValenceMapping__mdt	Valence Mapping	A mapping links a field in an external system to a field in this Salesforce organization.
valence__ValenceMappingFilterConfig__mdt	Valence Mapping FilterConfig	Junction record between Mapping and Filter that allows the Filter to store a configuration associated with this Mapping.

If you are building extensions, you will be manually creating Valence Adapter and Valence Filter records to represent your Apex classes.

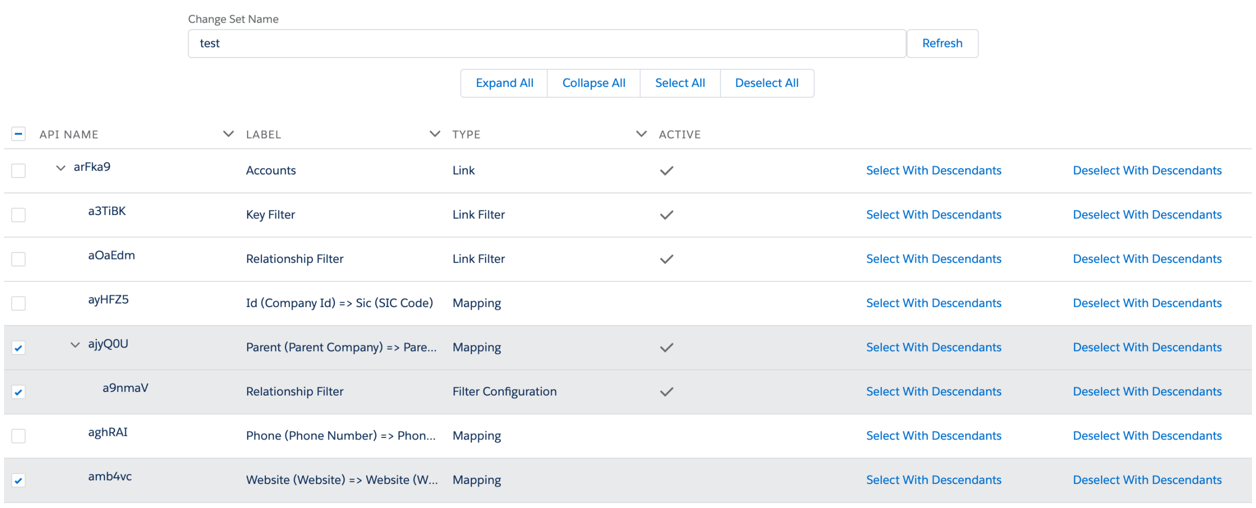
5.2.2 Valence Change Set Editor

The Change Set Editor allows you to add Valence configuration files to an existing Change Set. Recommended usage is to create an empty Change Set in the Setup Menu, then go into Valence to add entries to the Change Set.

Valence configuration records are naturally hierarchical. A Link has child Mappings and Filters, Mappings have child Configurations, etc. By working with the Change Set Editor inside of Valence you can see and work with this hierarchy, simplifying the process of building a Change Set that involves Valence entries.

For example, if you want to include a single Link and everything associated with that Link, you can click "Select With Descendants" on the Link row in the grid that you see.

Once you are satisfied with your changes, click the 'Save Changes' button to cause your selections to update the Change Set. Go back to the Setup Menu to deploy the Change Set to another Salesforce environment as you normally would.



Change Set Name

test

Refresh

Expand All | Collapse All | Select All | Deselect All

	API NAME	LABEL	TYPE	ACTIVE		
☐	arFka9	Accounts	Link	✓	Select With Descendants	Deselect With Descendants
☐	a3TiBK	Key Filter	Link Filter	✓	Select With Descendants	Deselect With Descendants
☐	aOaEdm	Relationship Filter	Link Filter	✓	Select With Descendants	Deselect With Descendants
☐	ayHFZ5	Id (Company Id) => Sic (SIC Code)	Mapping		Select With Descendants	Deselect With Descendants
✓	ajyQ0U	Parent (Parent Company) => Pare...	Mapping	✓	Select With Descendants	Deselect With Descendants
✓	a9nmaV	Relationship Filter	Filter Configuration	✓	Select With Descendants	Deselect With Descendants
☐	aghRAI	Phone (Phone Number) => Phon...	Mapping		Select With Descendants	Deselect With Descendants
✓	amb4vc	Website (Website) => Website (W...	Mapping		Select With Descendants	Deselect With Descendants

Note: Because Salesforce does not readily expose Change Sets for browsing and editing, we've had to do some creative work under the hood in order to create the Change Set Editor. The editor is stable and quite useful, but it does not have as many niceties as we would have liked to put in it.

- You cannot create a new Change Set, only modify an existing one.
- We cannot list existing Change Sets for you, you'll have to type in or paste the name.
- You cannot remove anything from a Change Set, only add to it. You can of course remove from the Setup Menu.
- You cannot upload your Change Set from here, you'll have to go back to the Setup Menu for that.

5.3 External Systems

When we say "external system", we're referring to something other than Salesforce that can participate in a data exchange. This might be a database in the cloud or on-premise. It might be a service you use, like Stripe or Quickbooks. Or maybe it's a messaging hub, like Kafka.

Data is exchanged with these external systems via **HTTPS**.

Interactions with external systems are handled by *Adapters*, and since these are Apex classes there is tremendous flexibility in how to converse with these external systems. SOAP or REST, XML or JSON. . . all are viable.

If you can reach it via HTTPS, you can talk to it with Valence.

5.3.1 Authentication

Valence uses the native Salesforce Named Credential feature to handle:

1. Defining URLs to reach external systems to interact with
2. Configuring authentication details and storing identity secrets

Named Credentials are a fantastic feature that allows a Salesforce admin to define an endpoint (URL) and authentication information to access that endpoint.

Valence leverages Named Credentials extensively to allow admins to easily define external systems they'd like to use Valence to exchange data with.

For a deeper dive on authentication, look at *Adapter Authentication*.

5.3.2 API Design

The more modern the API exposed by the external system, the simpler and easier a Valence Adapter will be to write. Features such as self-describing schemas, filtering based on timestamps, pagination, etc all contribute to a smooth build.

If you are responsible for designing and building the external API that Valence will talk to, take a look at our guide on *How to Design a Complementary Valence API*.

5.3.3 Systems Without APIs

Valence communicates with external systems using HTTPS, but sometimes you have a system that doesn't have an HTTPS API. Usually this is some kind of database (MySQL, SQL Server, Postgres, etc) being hosted somewhere, either in the cloud or on-premise.

For these scenarios, we recommend combining Valence with a great tool called SlashDB that wraps these sorts of database with an API layer. It's straightforward to set up, and there's already a prebuilt Valence adapter that works with SlashDB, so you can be up and running in no time.

5.4 Delta Sync

5.4.1 Overview

Typically when you are synchronizing data between two systems you have to move every record the first time (to fill up the target), and then from that point forward you prefer to only move things that have changed.

These smaller sips of data are called "delta" syncs, or sometimes "changes only" syncs. They are preferred because moving small amounts of data means faster and more efficient updates to the target system (and with Salesforce in particular, it's nice to have the `LastModifiedDate` field actually be meaningful and not have every record have the same value).

Delta syncing behavior is baked into Valence, and all source Adapters are expected to support it if the data they are talking to can be filtered appropriately.

Note: Sometimes, a data source can't accurately detect what has changed, and in this circumstance the fallback is simply to get all the records each time. Less efficient, but still quite effective.

Whenever a Link run occurs (generating a Sync Event), the timestamp at which that run occurred is stored on the Sync Event. Depending on the results of the run, that Sync Event is marked as successful or unsuccessful.

If successful, Valence will remember that timestamp, which we refer to as the "**last successful sync**" timestamp.

Note: Valence always runs allowing "partial success", meaning that if some records succeed they are allowed to be written to the target. A Sync Event is always marked strictly pass/fail, but a failed sync event might have 95% of records successfully written. We don't want to lose those stragglers, though, so for safety this is still considered a failed run overall.

During a Link run, the last successful sync timestamp is *made available* so that the source Adapter can use it with the data it is about to fetch. That source Adapter simply attaches a filter of "last modified >= lastSuccessfulSync" to whatever records it is gathering for the run.

Summary

By tracking the last successful sync and only updating it on a successful run, we ensure that no records are ever dropped on the floor and forgotten about because they had an error or there was an issue with the run. As an example, let's say you have a Link that ran daily. On Tuesday it is fetching all the records that have changed since Monday. Uh oh! There's a failure in the Tuesday run. It runs again on Wednesday, except this time it is fetching 48 hours of changes instead of 24, because the lastSuccessfulSync timestamp is still from the Monday run that was successful. For each failed run the window of records being fetched expands so that none are left behind.

This also means that Valence self-recovers from many kinds of typical integration issues. If your external system is offline, or there's a server error, etc, a run might fail but the next run will catch you up and things will be humming along again in no time. The combination of delta syncing and tracking the last successful timestamp is your first line of defense against integration issues. To learn about the others, check out *Handling Errors*.

Warning: A common mistake people make is to let one bad record "gum up the works". If you have a record that is malformed or broken in some way and causes a failed run every time it is included, the lastSuccessfulSync timestamp will not be advanced forward and your fetch window will get larger and larger over time. You might not notice right away if most of your records are individually still succeeding, as data will still be flowing, you'll just be missing some.

Solve this by keeping an eye on your Sync Events (checking from time to time, or setting up a Report or Dashboard or notification of your preference to monitor for Sync Event records where `valence__Success__c` == `false` and `valence__Status__c` == `Completed`).

5.4.2 Delta Sync With Cursors

Not every system uses a last modified timestamp to track changed records. Perhaps you're interacting with a message queue of some kind, where each message has a unique sequential number.

In these sorts of scenarios Valence still supports delta sync out of the box. Instead of working with *LinkContext*'s `lastSuccessfulSync`, use `lastSuccessfulCursor`. This is a value you tell Valence about, and then Valence gives it back to you on the next run. So, for example if you see as far as message 200, the next time you fetch records you will get "200" and can start from there. This doesn't have to be a numeric value, perhaps it's a GUID or some other identifier. What's important is that it makes sense to you and the system you are fetching data from.

Syntax for setting the cursor: `valence.RunContext.currentContext().setCursor(yourStringCursorValue);`

Tip: The same explanation above about successful and failed runs still applies. If you have run A that finishes at cursor 200 successfully, then run B starts at 200, finishes at 300 but is a failure, when run C starts it will receive "200", not "300" as its cursor value.

Example Usage

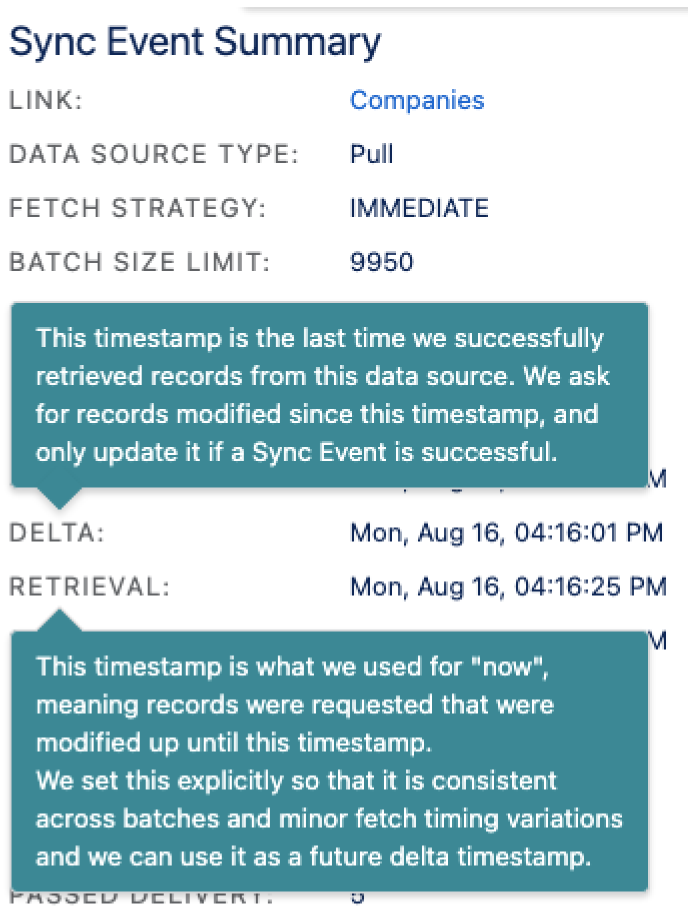
```
public List<RecordInFlight> fetchRecords(LinkContext context, Object scopeObject) {

        Integer messageNumber = 0;
        if(context.lastSuccessfulCursor != null) {
                messageNumber = Integer.valueOf(context.lastSuccessfulCursor);
        }

        List<Message> messages = fetchMessagesSince(messageNumber);
        List<valence.RecordInFlight> records = new List<valence.RecordInFlight>();
        for(Message message : messages) {
                records.add(buildRecordFromMessage(message));
                valence.RunContext.currentContext().setCursor(message.messageNumber);
        }

        return records;
}
```

5.4.3 Runs and Full Runs

Delta syncs are automatic and you don't have to do anything to keep them going. A few things to know:

Even though delta syncs are the default, the first run for a new Link always fetches all data (since it has no lastSuccessfulSync timestamp!). So typically your first run on a new Link gets everything, and then we grab small sips from there.

You can see the timestamp that was used as the start of the fetch window on the Sync Event Summary screen.



Sync Event Summary

LINK:	Companies
DATA SOURCE TYPE:	Pull
FETCH STRATEGY:	IMMEDIATE
BATCH SIZE LIMIT:	9950
DELTA:	Mon, Aug 16, 04:16:01 PM
RETRIEVAL:	Mon, Aug 16, 04:16:25 PM

This timestamp is the last time we successfully retrieved records from this data source. We ask for records modified since this timestamp, and only update it if a Sync Event is successful.

This timestamp is what we used for "now", meaning records were requested that were modified up until this timestamp.
We set this explicitly so that it is consistent across batches and minor fetch timing variations and we can use it as a future delta timestamp.

There are two buttons on the Link Dashboard screen that allow you to run the Link immediately.

1. **Run Now** - Do a normal Link run right now, which will be a delta sync
2. **Full Run** - Do a special link run right now that will ignore lastSuccessfulSync and fetch all records regardless of last modified date

We recommend doing a Full Run if:

- You've made changes to metadata or mappings and need fields from the source records you haven't fetched before
- You think you've had some data loss, or you've inadvertently deleted records from your target that you didn't want to
- Your business has some kind of data reconciliation process where they re-sync all their records on some schedule (quarterly, yearly, etc)

Note: If you don't see the **Run Now** and **Full Run** buttons on the Link Dashboard screen, it's because this Link uses a data source that does not fetch records. Some Links are designed to accept real-time record pushes and don't control when or how they receive data.

5.5 Testing Mode

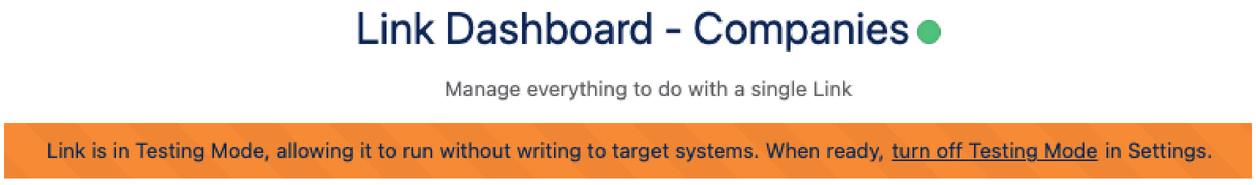
During development of a new Link you are usually working with test environments and can write data wherever without causing much fuss.

Sometimes, though, it's important that you not make changes to a target system you are building a Link against until the Link is closer to being finished. That's where Testing Mode comes in.

Testing Mode is toggled on or off on a Link-by-Link basis from the Link Settings screen (the same place where you rename a Link or change its Logging Level).

When turned on, all Adapters and Filters *can see a flag* that indicates that this Link run is a test mode run so they can behave accordingly. Typically this only matters for the target Adapter so that it knows not to persist data.

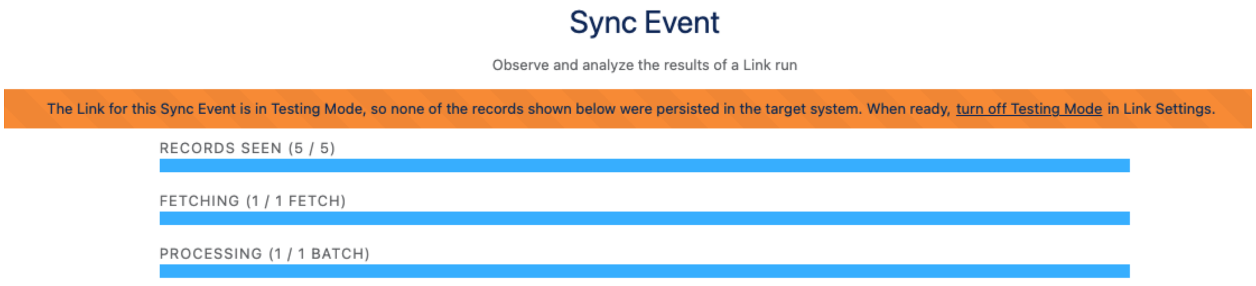
When Testing Mode is on for a Link, you'll see a banner on the Link Dashboard:


Link Dashboard - Companies ●

Manage everything to do with a single Link

Link is in Testing Mode, allowing it to run without writing to target systems. When ready, turn off Testing Mode in Settings.

You'll also see a banner on the Sync Event Summary screen:


Sync Event

Observe and analyze the results of a Link run

The Link for this Sync Event is in Testing Mode, so none of the records shown below were persisted in the target system. When ready, turn off Testing Mode in Link Settings.

Note: Target Adapters vary in sophistication for what they can still do in Testing Mode. Most Adapter simply won't try to send records, so even though you're not seeing errors it doesn't mean what the Link is producing would be accepted by the end system. This is why we encourage you to only use Testing Mode if you need to, otherwise use a destination environment you are comfortable writing data to.

As an example of what's possible, when you are writing in Salesforce in Testing Mode, you will actually see real database error messages (such as a required field being missing, or a validation rule failing). This is because Salesforce supports database transaction rollbacks, meaning we can actually write, see errors, then discard the pending write.

5.6 Handling Errors

5.6.1 Overview

All integrations fail at some point. It's not possible to have so many moving parts and interconnected systems and not have something go awry.

What's important is how we handle it.

Here's our philosophy when it comes to integration error handling:

- At a bare minimum we should know that something went wrong.
- Beyond knowing something is wrong, it would be nice to have a sense of what specifically went wrong, and perhaps why, and some clues towards a resolution (the more detail the better).
- Beyond knowing the details of the failure, it would be good if the system could recover on its own (if possible).
- If it can't recover on its own, then there should be a way that we can help it recover manually.

These principles are reflected in the error handling and recovery designs within Valence, which are battle-tested and quite resilient.

5.6.2 Types of Errors

Let's define a few different patterns of errors that can occur so we can talk about what each means and how to deal with them.

Errors are captured by Valence and shown on the Sync Event Summary screen in realtime while the Link is running, and also afterwards.

System Error

This kind of error happens at a very fundamental part of a Link run. Maybe the other external server is offline, or we have the wrong authentication info, etc.

System errors stop the Link run. Things are too broken to continue (or sometimes, to even start).

Batch Error

Valence is highly-parallel and splits the work into a number of batches, each in its own execution context. A batch error fails an entire batch, but the overall Link run continues.

Maybe when Valence goes to fetch a particular page of results one of those results is malformed and that page throws an HTTP 400 error (saw that one just recently). Batch errors show at the top of the Sync Event Summary screen and also within each batch.

Record Error

A single record has an error that is blocking its individual success (and consequently the success of the overall job). The most common cause of these are things like a record has a blank field that is required in the other system, or its value is too long, or fails validation in some way. Every database has some bad data and you'll find yours with Valence.

This record will not succeed but its siblings in the batch might, and certainly other batches will succeed or fail regardless of this record's plight.

When a record error occurs Valence will take a snapshot of the state of that record before and after transformations, and you will be able to browse and look at field values for all failing records. This is one of our most powerful features and is hugely beneficial in diagnosing issues.

Honorable Mention: Record Warning

Not technically an error but worth mentioning here, a warning is applied to a record but does not block its success. This is a way to flag things for attention that aren't worth failing a run over.

Honorable Mention: Ignore Reason

Separate from a warning, a record can be marked as ignored by any Adapter or Filter. This is commonly used to filter or refuse records that you are receiving but know you don't want to pass along to the target system. When records are marked in this way the Adapter or Filter that did so is required to provide a reason, and that reason is shown on the SyncEventSummary screen.

5.6.3 Error Durability

In addition to types of errors, we're also going to separate errors into two categories based on duration:

1. Ephemeral - short-lived, likely to go away if circumstances were a little bit different
2. Durable - long-lived, something intrinsic to the data or the configuration

An ephemeral error is something like an external server being temporarily unavailable, but if we tried it again in a couple minutes or a couple hours it would work. (One of our customers has an API that seems to take a nap every night at midnight for a little bit, so their hourly run fails exactly once a day.) Another example would be some kind of record locking contention because of parallel writing.

A durable error is typically based on the data itself, or how the Link is configured. If we have the wrong authentication details, trying again in an hour isn't going to make that go away. If we have a record with a malformed email address, we can't keep jamming it into Salesforce hoping it will be accepted.

5.6.4 Valence Error Handling

Now that we've laid out ways of categorizing and labeling errors, let's walk through what features exist in Valence to help us mitigate or resolve these errors.

Last Successful Sync

If you've read our explanation of *Delta Sync*, then you're already familiar with this concept. This is your main line of defense against integration errors. If a run fails, the same records (plus possibly some additional data) will be tried again on the next run.

Let's talk about how this helps us with our types of errors:

1. **Durations**
    1. *Ephemeral* - great; we'll be trying the same data again on the next run
    2. *Durable* - good; can't resolve durable errors by itself, but if you fix the problem (repair a record, change the auth credentials, etc) the next run will take advantage of your work so you can just correct and sit back, no further work needed.
2. **Types**
    1. *System Error* - great; since we end up with a fresh run, these are discarded (of course the new run might throw more)
    2. *Batch Error* - great; same as above
    3. *Record Error* - great; same as above
3. **Modes**
    1. Pull/Fetch - Supported
    2. Push/Realtime - Not Supported

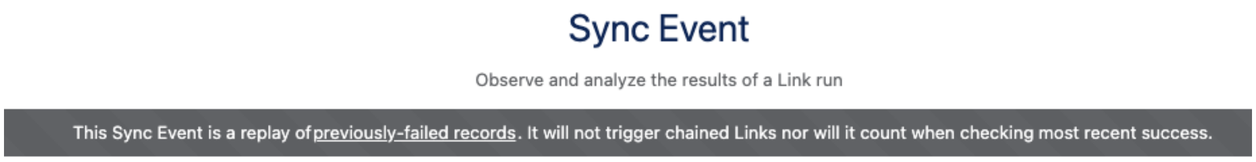
Note: Be aware that Last Successful Sync only applies to Links that are pulling data from somewhere (like a scheduled Link fetching invoices hourly from Quickbooks). We can tell that source Adapter "Hey, give me those same 1000 records again please". If our Link is being fed data in realtime (such as from an Apex trigger or from an external system sending records into Valence via the Apex REST API), then we have no way to go back to the well to get records again. We'll talk about this scenario more in *Retry Failures*.

5.6.5 Automated Replay

There is a feature in the Settings screen for a Link that can be toggled on and off (default: off), and you can also configure a maximum number of attempts.

Automated replays exist for those scenarios that are *very* ephemeral, where a record is failing but the same exact record just seconds or minutes later would not have an error. We added this feature to help people with highly-concurrent inbound to Salesforce record flow, especially with realtime pushes coming in from external systems via the Apex REST API.

If you have automated replays turned on and some records in a run fail, just those records are immediately processed in a new Link run, and that run will attempt itself repeatedly until either it hits the attempts limit or all records are successful. Replay Sync Events will have a banner like this:



Sync Event

Observe and analyze the results of a Link run

This Sync Event is a replay of previously-failed records. It will not trigger chained Links nor will it count when checking most recent success.

1. **Durations**

    1. *Ephemeral* - great; designed for very short duration ephemeral errors
    2. *Durable* - N/A

2. **Types**
    1. *System Error* - N/A (only re-processes failed records)
    2. *Batch Error* - N/A (only re-processes failed records)
    3. *Record Error* - great; designed to fix record-level errors, will grab individual failing records and keep trying them again (and if some succeed it will only keep retrying the failures)
3. **Modes**
    1. Pull/Fetch - Supported
    2. Push/Realtime - Supported

5.6.6 Retry Failures

Retry failures is an operation you can trigger manually from the Sync Event Summary screen after a Link run is over if it had errors of any kind. This starts a new Link run with some unique behavior.

1. Any batches that threw errors and failed outright (no records even received for that batch) will be retried (fully successful batches will not be retried)
2. Any individual failing records whose batch finished (didn't explode) will also be retried (we call these "loose failing records" since they are gathered up from all the batches).

This is a very powerful tool. Let's say you have a big run that pulls in three million records, but 15 batches timed out during the fetch, and you also had a validation rule you forgot about block about 70,000 records of the 3,000,000. You go in and fix the validation rule and you're ready to get your missing records. Since the run overall was a failure you *could* use **last successful sync** to pull everything again, but that's a lot of records you already successfully received.

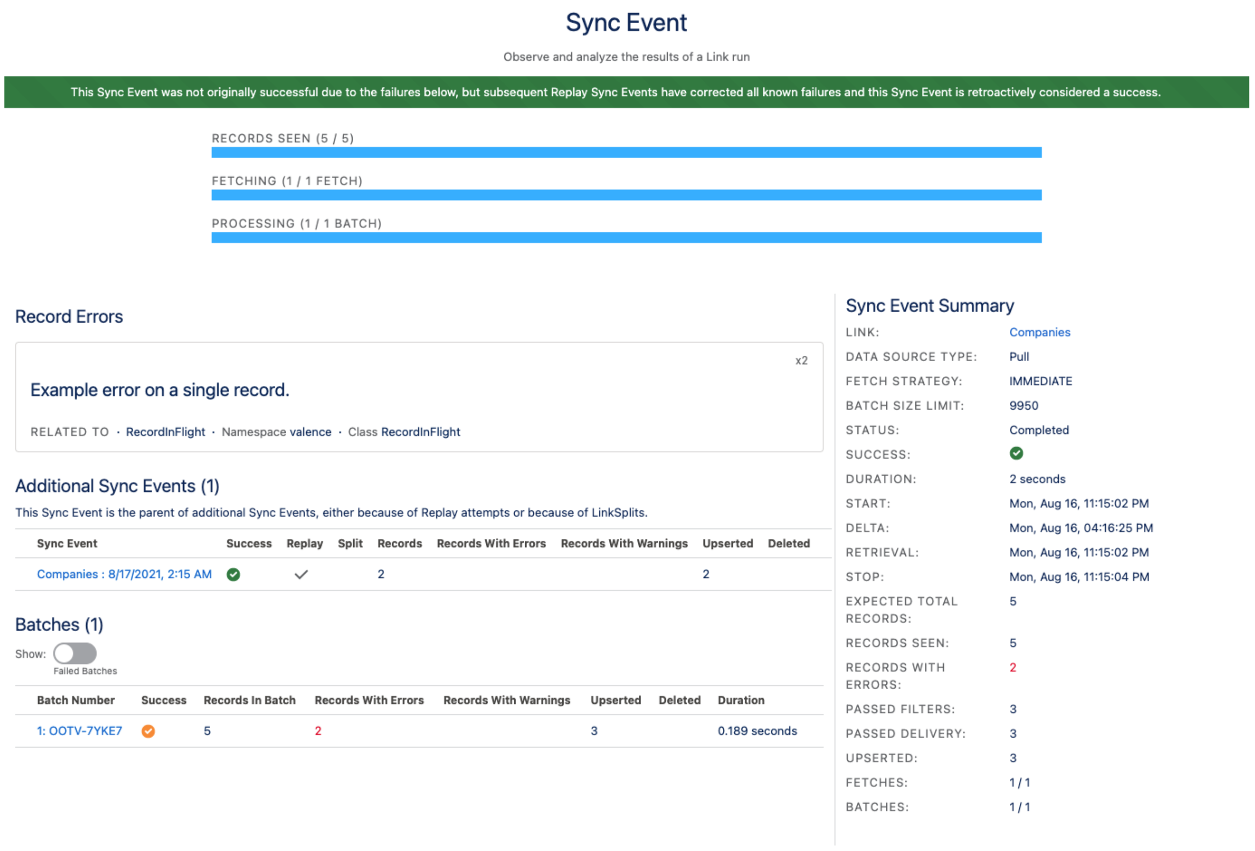
So, instead of doing another normal run you click "Retry Failures". A new Link run starts with, say, 30 batches. That's the 15 failing batches, plus enough batches to put all the loose failing records into batches of their own for reprocessing. Way fewer records, so this run is done really quickly. We fixed the validation rule so those 70,000 records came in successfully, and out of our 15 bad batches 11 of them were successful. But…uh oh…four batches timed out again! Shoot.

What do we do? Click **Retry Failures** on this replay Sync Event, of course! You can chain retries as many times as you need to.

This third run (2nd retry) is *even faster* because it's just four batches, and thankfully all four come through and we've finished our three million record run. Congrats!

All of the replay runs will link up to the original failing Sync Event, and likewise that original Sync Event will list the child replay attempts on its Sync Event Summary screen.

And, if we are eventually fully successful through our replay attempts, *the original Sync Event is changed from a failure to a success*!



Sync Event

Observe and analyze the results of a Link run

This Sync Event was not originally successful due to the failures below, but subsequent Replay Sync Events have corrected all known failures and this Sync Event is retroactively considered a success.

RECORDS SEEN (5 / 5)

FETCHING (1 / 1 FETCH)

PROCESSING (1 / 1 BATCH)

Record Errors

x2

Example error on a single record.

RELATED TO · RecordInFlight · Namespace valence · Class RecordInFlight

Additional Sync Events (1)

This Sync Event is the parent of additional Sync Events, either because of Replay attempts or because of LinkSplits.

Sync Event	Success	Replay	Split	Records	Records With Errors	Records With Warnings	Upserted	Deleted
Companies : 8/17/2021, 2:15 AM	✓	✓		2			2	

Batches (1)

Show: Failed Batches

Batch Number	Success	Records In Batch	Records With Errors	Records With Warnings	Upserted	Deleted	Duration
1: OOTV-7YKE7	✓	5	2		3		0.189 seconds

Sync Event Summary

LINK:	Companies
DATA SOURCE TYPE:	Pull
FETCH STRATEGY:	IMMEDIATE
BATCH SIZE LIMIT:	9950
STATUS:	Completed
SUCCESS:	✓
DURATION:	2 seconds
START:	Mon, Aug 16, 11:15:02 PM
DELTA:	Mon, Aug 16, 04:16:25 PM
RETRIEVAL:	Mon, Aug 16, 11:15:02 PM
STOP:	Mon, Aug 16, 11:15:04 PM
EXPECTED TOTAL RECORDS:	5
RECORDS SEEN:	5
RECORDS WITH ERRORS:	2
PASSED FILTERS:	3
PASSED DELIVERY:	3
UPSERTED:	3
FETCHES:	1 / 1
BATCHES:	1 / 1

1. **Durations**
   1. *Ephemeral* - N/A - use one of the other strategies
   2. *Durable* - great; since it will not run until you tell it to, and you can try it as many times as you want while you iterate through your fixes and troubleshooting
2. **Types**
   1. *System Error* - N/A (only re-processes failed batches and failed records)
   2. *Batch Error* - great; will retry any batches that failed in the previous run
   3. *Record Error* - great; will cherry-pick failing records from each batch of previous run and reprocess them in the new run
3. **Modes**
   1. Pull/Fetch - Supported (see additional node)
   2. Push/Realtime - Supported

A few extra things to be aware of:

- If you set a Link's "logging level" in the Settings screen to "None", you will not be able to retry individual records since no snapshot of them would have been saved. Set to any level other than "None" to capture failing records.
- This is the only mechanism where you can retry records that were pushed in via a realtime data source. That's because Valence captures a record "snapshot" and serializes it, so if we need to reprocess that record we have a copy to work with and don't need to go back to the source.
- With the high value from the previous point comes a warning: once a snapshot has been captured it's going to start going stale. If you retry a record you are always running the risk that you will overwrite good data if that

record has been updated in the time from the original failure and when you click retry. Typically this is not an issue but just take a minute before you click Retry Failures and think about how or if these records might have already come through again successfully.

5.7 Invoking Valence

"Invoking Valence" means, "causing Valence to run", typically programmatically.

Most of the time you will configure Valence Links on a schedule or chain, and let them hum away on their own.

There are other uses cases, such as realtime runs, where you will need to kick off Link runs in a different way. Here are your options.

5.7.1 Process Builder or Flow

Salesforce supports something called Invocable Apex, which allows a bit of Apex code to be an Action within a process or flow. Valence supports this mechanism with two ways you can invoke a Link run:

1. Start a Link run that will fetch its own records

InvocableValencePull (Label='Run Valence Pull Link' Description='Kick off a Valence Link.')

Accepts a `List<Request>`, where Request looks like:

```
global class Request {

        @InvocableVariable(Label='Link Name' Description='The Link to run' Required=true)
        global String linkName;
}
```

2. Start a Link run with specific records

InvocableValencePush (Label='Run Valence Push Link' Description='Kick off a Valence Link that will process the passed records.')

Accepts a `List<Request>`, where Request looks like:

```
global class Request {

        @InvocableVariable(Label='Link Name' Description='The Link to run' Required=true)
        global String linkName;

        @InvocableVariable(Label='Record ID' Description='A record for the Link to␣
↪process' Required=true)
        global Id recordId;
}
```

5.7.2 Apex REST API

An external system that has Salesforce User credentials can call into Valence using the Salesforce REST API to drop off data for Link processing. This is often used in realtime push-style Links where an external system is emitting records to Valence as they are modified or created.

This is done with a HTTPS POST being sent to a unique URL for each Link, which is in the format:

`https://<org_base_url>/services/apexrest/valence/link/v1/<link api name>`

You can read more about this method in our guide *Sending Realtime Records To Salesforce*

Note: In order to accept records like this, the source Adapter on the Link must implement *SourceAdapterForRawDataPush*.

5.7.3 Apex

You can invoke a Link programmatically from Apex a number of different ways using the `valence.LinkEngine` global class.

```
/**
 * Attempt to run a Link in this same execution context. This only works for Links that␣
↪do not perform any callouts, as
 * a SyncEvent record will be created before calling Adapters, and DML before callouts␣
↪causes an exception.
 *
 * @param linkName Unique identifier for the Link that should be run (Must be a␣
↪ValenceDataLink__mdt.QualifiedAPIName)
 * @throws LinkException if the Link cannot be found or is not the right type of Link␣
↪for this method
 */
global static void runLinkSynchronously(String linkName);

/**
 * Attempt to run a Link. This only works for Links that use a Source Adapter that knows␣
↪how
 * to fetch its own data (a Pull).
 *
 * @param linkName Unique identifier for the Link that should be run (Must be a␣
↪ValenceDataLink__mdt.QualifiedAPIName)
 * @throws LinkException if the Link cannot be found or is not the right type of Link␣
↪for this method
 */
global static void runLink(String linkName);

/**
 * Attempt to run a Link. This only works for Links that use a Source Adapter that knows␣
↪how
 * to fetch its own data (a Pull).
 *
 * This version of the method ignores Valence's normal behavior of using Sync Event␣
↪timestamps to only fetch deltas.
```

(continues on next page)

(continued from previous page)

```
 *
 * @param linkName Unique identifier for the Link that should be run (Must be a␣
↪ValenceDataLink__mdt.QualifiedAPIName)
 * @throws LinkException if the Link cannot be found or is not the right type of Link␣
↪for this method
 */
global static void runFullLink(String linkName);

/**
 * Attempt to immediately run a Link. This method only works for Links that use a Source␣
↪Adapter that knows
 * how to consume SObject records (a Source Adapter that implements␣
↪SourceAdapterForSObjectPush).
 *
 * @param linkName Unique identifier for the Link that should be run (Must be a␣
↪ValenceDataLink__mdt.QualifiedAPIName)
 * @param records Some SObject records that will be fed to the Link for processing
 * @throws LinkException if the Link cannot be found or is not the right type of Link␣
↪for this method
 */
global static void runLink(String linkName, List<SObject> records);

/**
 * Attempt to immediately run a Link. Bypasses the Source Adapter and immediately starts␣
↪processing
 *
 * @param linkName Unique identifier for the Link that should be run (Must be a␣
↪ValenceDataLink__mdt.QualifiedAPIName)
 * @param records Some RecordInFlight records that will be fed to the Link for processing
 * @throws LinkException if the Link cannot be found or is not the right type of Link␣
↪for this method
 */
global static void runLink(String linkName, List<valence.RecordInFlight> records);

/**
 * Attempt to immediately run a Link. This method only works for Links that use a Source␣
↪Adapter that knows
 * how to consume SObject records (a Source Adapter that implements␣
↪SourceAdapterForRawDataPush).
 *
 * @param linkName Unique identifier for the Link that should be run (Must be a␣
↪ValenceDataLink__mdt.QualifiedAPIName)
 * @param recordData Some raw data that will be fed to the Link for processing
 * @throws LinkException if the Link cannot be found or is not the right type of Link␣
↪for this method
 */
global static void runLink(String linkName, String recordData);
```

Here are some ways you might invoke Valence from Apex:

- In an Apex trigger so you can do realtime outbound record flows from Salesforce
- In reaction to a user action, such as clicking a button in a custom app in your own interface

- As part of an automated process of your own, such as processing job that sends records out to an external system at the end of the job

5.8 Release Notes

5.8.1 2.10

Installation URL

Date: March 17, 2025

- Update to API 63
- Add global Link Run API that accepts a list of RecordInFlight
- Optimzie internal batch info aggregation
- SyncEventPruner to leave at least one successful Sync Event to improve delta timestamp calculation
- Valence no longer allows replaying a Sync Event that didn't collect Record Shapshots
- Change away from VF Remoting for dashboard data retrieval
- Salesforce Value Groomer improves handling for empty strings in non-string target field types

5.8.2 2.9.2

Date: Dec 12, 2024

- Bugfix: Improve Salesforce target name conflict management when pushing records

5.8.3 2.9.1

Date: Dec 10, 2024

- Bugfix: Correct Relationship Filter when multiple relationships exist on the same object but different fields
- Bugfix: Correct Record Snapshot view when there are records from more than 2000 batches
- Bugfix: Improve backfilling relationships after parent record is added but fields are different types

5.8.4 2.9

Date: Sep 30, 2024

- Allow Valence to be installed in orgs with any standard fields encrypted
- Enable table preferences such as number of records or filters to persist in browser
- Add Valence Configuration record to allow scratch and trial orgs to go beyond max queueable depth of 5
- Bugfix: Removing one side of a mapping resulted in a UI error message
- Bugfix: Mapping wizard didn't mark field usage on first display
- Bugfix: Invalid field paths could result in corrupted UI artifacts in Transformation view

5.8.5 2.8

Date: August 9, 2024

- Record Snapshot view can now search properties and messages †
- Bugfix: Local Salesforce adapter configuration fails to load

† Manual Upgrade Step: Valence customers upgrading from prior to 2.8 will need to manually edit the Valence Record Snapshot object and check "Allow Search" to enable the improved search results.

5.8.6 2.7.1

Date: July 9, 2024

- Bugfix: Record Snapshot Grid may not preserve Sync Event filtering
- Bugfix: Record Snapshot Grid can crash on invalid JSON properties

5.8.7 2.7

Date: July 8, 2024

- Bugfix: Resolve NoErrorObjectAvailable error thrown when viewing sync event.
- Bugfix: Change interface evaluation strategy, failing due to Summer '24 bug.
- Bugfix: Mapping editor doesn't reset filtering when clicking away.
- Bugfix: Large number of problems in a batch prevents the batch from resolving.
- Group problem summaries by generic exception message.
- When needed, compact problem summaries to fit Sync Event field space.
- Update Record Snapshot grid, move properties view to modal.
- Support IMF-fixdate/RFC7231 for inbound date format.
- Mapping editor now visually indicates if a property is already used in another mapping.
- Update SOQL and DML security enforcement.
- Update to API version 61 (Summer '24).

5.8.8 2.6.1

Date: March 20, 2024

- Bugfix: Resolve navigation break caused by enabling browser back and forward buttons.

5.8.9 2.6

Date: March 14, 2024

- Bugfix: Resolve an issue with the save button on Adapter configurations remaining disabled when changing Named Credential.
- Update to how Valence selects changed Salesforce records to catch potential misses. More work on this forthcoming.
- Browser back and forward buttons will now affect the Valence UI.
- UI improvements in Mappings and initial page load experiences.
- Updated to API version 60 (Spring '24).

5.8.10 2.5

Date: January 12, 2024

- PlanFetchAgain() will now always execute in an asynchronous context (higher limits), previous it was sync / async depending on how DELAY was called.
- Expanded checks for errors when publishing Platform Events, will show up in Valence Logs if they occur.
- Bugfix: Resolve an edge case with some exceptions that would cause an overcount in a Sync Event's jobsCompleted value.
- Bugfix: Resolve a rare race condition that would sometimes close a Sync Event too early.
- Bugfix: Resolve a rare race condition that would sometimes mark a Sync Event as successful despite a failing batch.
- Bugfix: Resolve an edge case with planFetch() that would sometimes fail to immediately close a Sync Event.
- Bugfix: Resolve an edge case with CUMULATIVE_SCOPES that would sometimes fail to immediately close a Sync Event.
- Increase the maximum number of encryptable bytes to handle serializing large records.
- Update ValueConverter to correctly parse millisecond timestamps stored as string values when writing to DateTime fields.
- Bugfix: Truncate recent values saved for records to avoid exceeding the maximum number of storable characters.
- Updated documentation to improve clarity.

5.8.11 2.4

Date: December 4, 2023

- Bugfix: Resolve a race condition with parallel batches that sometimes would leave a Sync Event in an incomplete state.
- Updated documentation in a number of places to improve clarity.

5.8.12 2.3

Date: October 25, 2023

- Update to Salesforce API version 59.0 (Winter '24).

5.8.13 2.2.1

Date: September 28, 2023

- Bugfix: Correct a key matching edge case in RelationshipFilter introduced in 2.2.
- Bugfix: Correct an issue with ChildDetective throwing errors about quotes in some circumstances.

5.8.14 2.2

Date: September 5, 2023

- Add missing documentation on FieldBuilder.addChild().
- New feature allowing Full Run mode to toggled on and off per-Link, allowing scheduled Links to fetch all data every time if desired.
- Bugfix: correct an issue with numeric Salesforce fields used as keys when upserting into Salesforce, or doing relationship lookups.
- Bugfix: better handle keys with unusual characters when upserting into Salesforce, or doing relationship lookups.

5.8.15 2.1

Date: April 18, 2023

- Add the ability to store *Additional Table Details*, useful for Adapters with more complex table definitions.

5.8.16 2.0.4

Date: February 11, 2023

- Expand the SF Value Groomer filter to handle a wider variety of value formats when writing to Time fields in Salesforce.

5.8.17 2.0.3

Date: January 23, 2023

- Bugfix: fix a rare issue with an internal tracking value in high-volume orgs.

5.8.18 2.0.2

Date: January 19, 2023

- Bugfix: fix an issue with the ChangeSet editor no longer parsing API responses as XML.

5.8.19 2.0.1

Date: November 17, 2022

- Bugfix: fix a display issue on the Subscription Details screen when Lightning Web Security is enabled.
- Bugfix: resolve a NullPointerException being thrown in certain circumstances of parallel batch processing.

5.8.20 2.0

Date: October 27, 2022

- Update to Salesforce API version 56.0 (Winter '23).
- Bugfix: avoid a javascript library issue when Lightning Web Security is enabled (LWS enabled by default in new orgs starting Winter '23).
- Bugfix: certain UI screens show a javascript access denied error, suspected Winter '23 side effect.

5.8.21 1.102

Date: October 21, 2022

- Add a sequence number to every batch that is processed so they can be arranged independently of the order in time where they are created.
- When using custom fetch cursors, use the new sequence numbers to set the cursor to the one from the "last" batch (even if it was not the last batch processed).
- More detailed stored info about the configured schedule for a Link:
  - Clearer explanation in the UI for what the schedule is.
  - Opening the schedule UI sets the form to the current configured values.
  - New feature: a warning if a Link has a saved schedule that is not active, with the ability to click to restore the schedule.
  - These new expanded behaviors will only be present on Links scheduled after this update is installed; reschedule older Links if you want the new features on them.
- Expand tracking and display of custom fetch cursors.
- Bugfix: UI exception in a specific circumstance should block movement to next screen.
- Performance improvement: one UI screen was not preloading its assets.

5.8.22 1.101

Date: August 1, 2022

- Change how empty batches are marked and evaluated.
- Bugfix: incorrect sync event tracked value in certain circumstances when using the CUMULATIVE_SCOPES FetchStrategy.

5.8.23 1.100

Date: May 21, 2022

- Bugfix: Links scheduled to run every X minutes that take longer than X minutes for a run stop re-scheduling.
- Delta syncing now has an additional mode that Adapters can use called *Delta Sync With Cursors*.
  - Typical delta sync is done using a timestamp and a last modified field, but some systems will use a different pattern.
  - This new feature allows Adapters to use their own custom cursor to track which records they want to fetch. Adapters can hand values to Valence for storage, and receive them back on future runs.
- Update some of the documentation to correct typos and bring syntax up-to-date.

5.8.24 1.99.1

Date: April 25, 2022

- Bugfix: Links scheduled to run every X minutes that take longer than X minutes for a run stop re-scheduling.

5.8.25 1.99

Date: January 24, 2022

- Bugfix: incorrect help popup positioning in a few different UI screens.
- Bugfix: regression where the max automated replay configured value is not respected when Automated Replays are turned on for a Link.

5.8.26 1.98

Date: December 3, 2021

- Bugfix: correct an issue in the Change Set Editor UI screen.
- Bugfix: subtle bug in the delta timestamps used in remote Salesforce record queries.
- Bugfix: mapping drill-down button doesn't work for nested object fields.
- Bugfix: wrong default selector when starting a mapping from a target list field.
- Bugfix: ugly formatting for configuration explanation when LocalSalesforceAdapter has custom SOQL added then removed.
- Bugfix: Exception in **ValenceUtil.createTestMapping()** when target side is null.
- Bugfix: Mappings using advanced mode list selectors should open in advanced mode.

- Bugfix: Parallel batch runs don’t respect the isFullRun flag.

5.8.27 1.97

Date: November 11, 2021

- Allow mappings to be configured per-mapping to ignore or persist null values. If ignored, an incoming null value will not overwrite an existing value.
- Pop a warning in the Valence UI if the current user does not have the **Valence User** permission set assigned to them.

5.9 Schema

5.9.1 What Does a Schema Look Like?

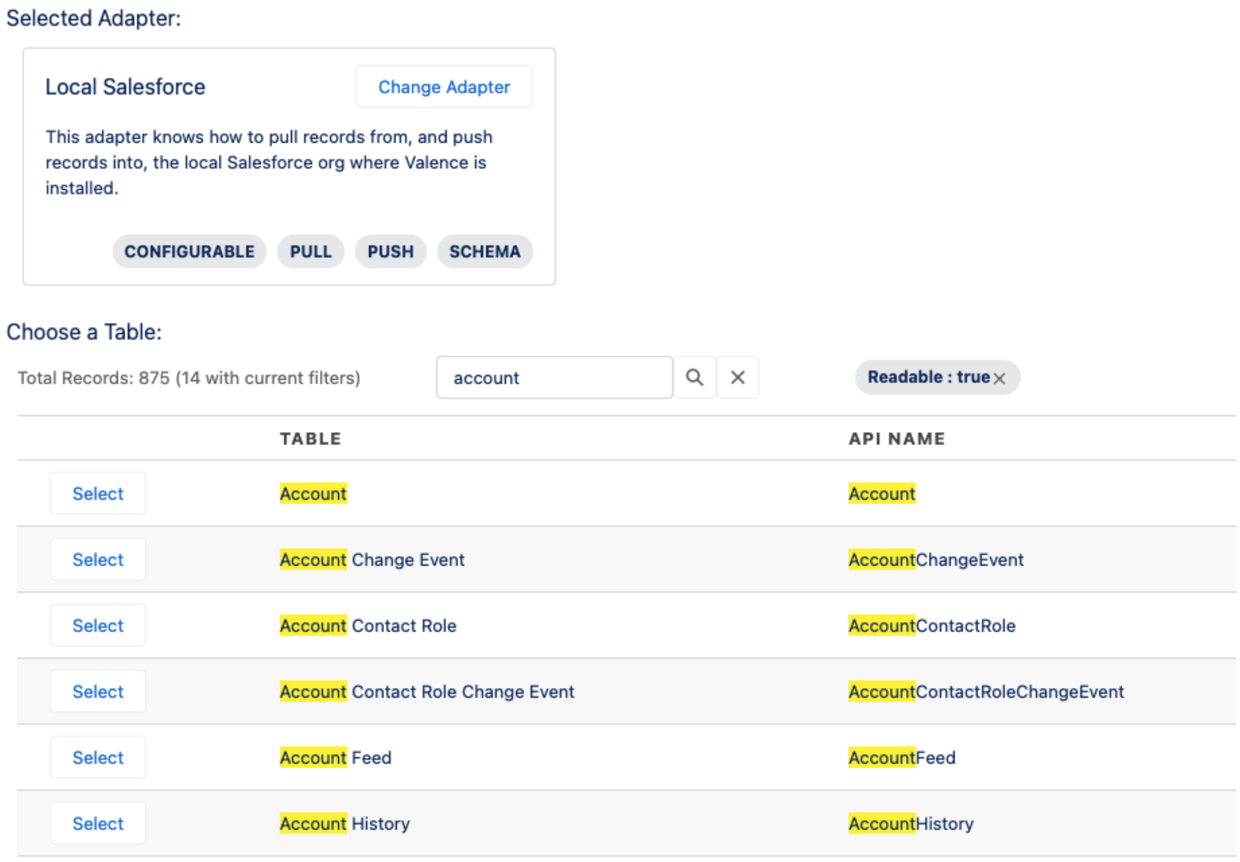
Schemata are formal descriptions of the data modeling of records. We also sometimes refer to this as the record’s “shape”. They answer questions like:

- What entities are available to read from? To write to?
- What fields exist on those entities?
- What’s the data type of each field? Restrictions on it? (character length, required, etc)
- What are the relationships to other entities? What is their cardinality? (One to Many, One to One, etc)

Valence has a sophisticated and nuanced understanding of schemata. It’s a integral part of the engine, since the primary purpose of any middleware is to translate data from one schema to another!

Table Schema

When you are setting up a Link you are presented with your options for which table (a.k.a. entity, object) to interact with. This is a dynamic investigation by Valence and will reflect any changes in the schema that have occurred (for example, a new custom object being added to your Salesforce org). You can search and filter to help narrow your selection. Also, this screen is context-aware; for example, when selecting a target table, any tables that came back as not-writeable won't be shown.


Selected Adapter:

Local Salesforce

This adapter knows how to pull records from, and push records into, the local Salesforce org where Valence is installed.

Choose a Table:

Total Records: 875 (14 with current filters)

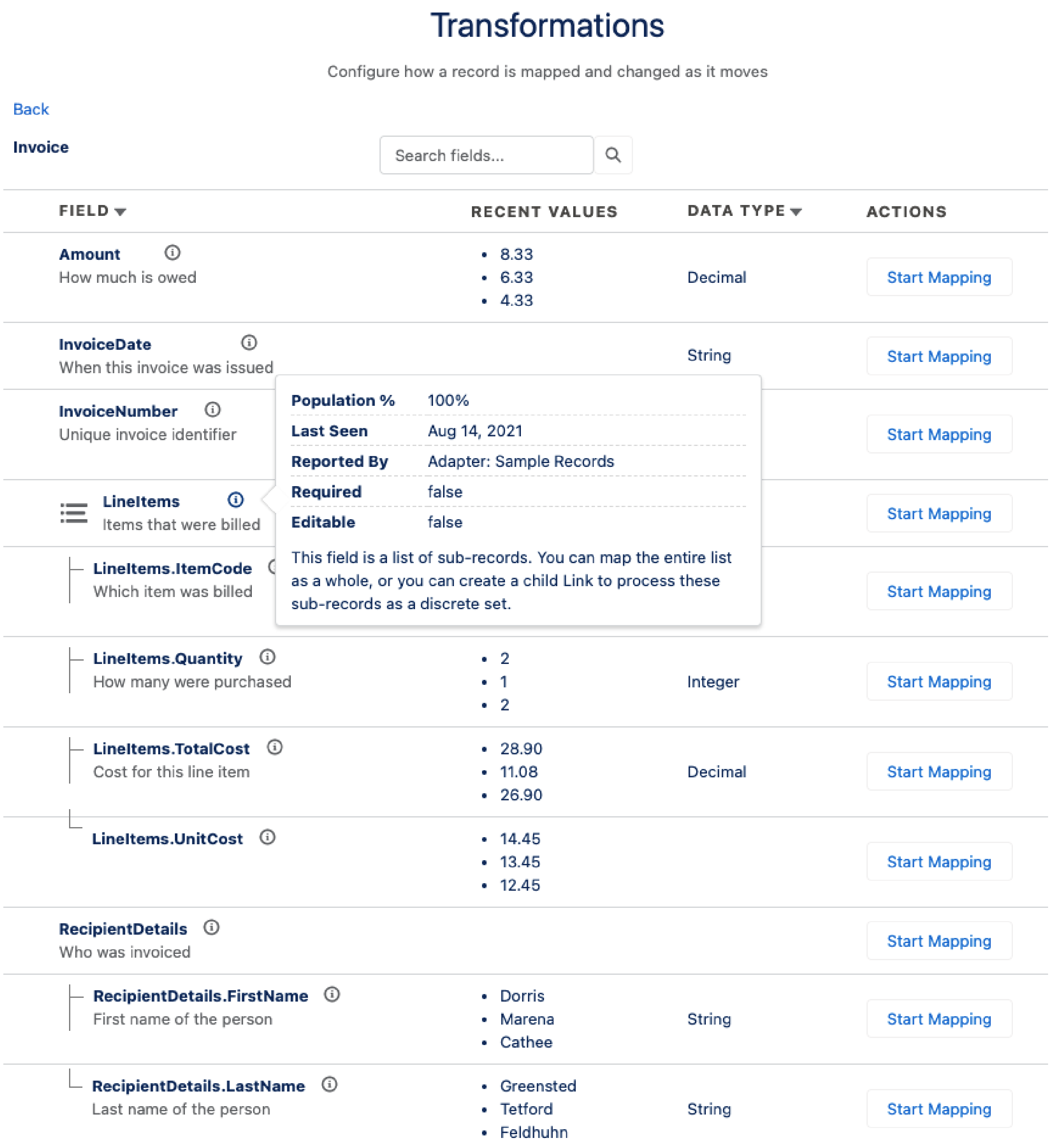
account

Readable : true

	TABLE	API NAME
Select	Account	Account
Select	Account Change Event	AccountChangeEvent
Select	Account Contact Role	AccountContactRole
Select	Account Contact Role Change Event	AccountContactRoleChangeEvent
Select	Account Feed	AccountFeed
Select	Account History	AccountHistory

Field Schema

Each record has two schemata: what it looks like in the data source system, and what it should look like in the target system. Valence provides a schema browser as part of configuring a Link that will allow you inspect and develop an understanding of these record shapes.



Transformations

Configure how a record is mapped and changed as it moves

Back

Invoice

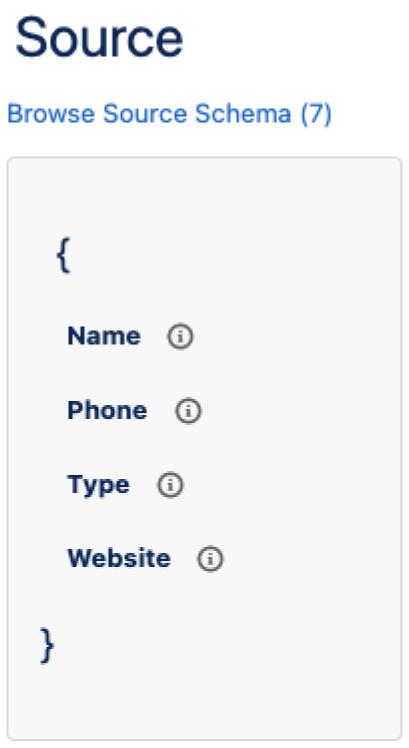
Search fields...

FIELD	RECENT VALUES	DATA TYPE	ACTIONS
Amount How much is owed	8.33, 6.33, 4.33	Decimal	Start Mapping
InvoiceDate When this invoice was issued		String	Start Mapping
InvoiceNumber Unique invoice identifier			Start Mapping
LineItems Items that were billed			Start Mapping
LineItems.ItemCode Which item was billed			Start Mapping
LineItems.Quantity How many were purchased	2, 1, 2	Integer	Start Mapping
LineItems.TotalCost Cost for this line item	28.90, 11.08, 26.90	Decimal	Start Mapping
LineItems.UnitCost	14.45, 13.45, 12.45		Start Mapping
RecipientDetails Who was invoiced			Start Mapping
RecipientDetails.FirstName First name of the person	Dorris, Marena, Cathee	String	Start Mapping
RecipientDetails.LastName Last name of the person	Greensted, Tetford, Feldhuhn	String	Start Mapping

Population %	100%
Last Seen	Aug 14, 2021
Reported By	Adapter: Sample Records
Required	false
Editable	false

This field is a list of sub-records. You can map the entire list as a whole, or you can create a child Link to process these sub-records as a discrete set.

As you build up mappings and transformations, Valence will also show you a representation of your record as it is currently being used (only showing which fields are being read, or written to, etc) so you can have a really crisp understanding of what you're actually working with.



5.9.2 Where Does the Schema Come From?

The schema you see in the schema browser is actually built up from many layers of information that Valence can access that each provide clues about record shape. What you see is the distillation of all of these layers into what the record truly looks like, and why.

We do this because APIs can be unreliable when describing what their entities look like. Maybe it says it will send you seven fields but actually it sends nine. If all we did was trust the API's declaration, you'd never be aware of (least of all, use) those extra fields.

1. Adapters

Adapters can share information about their schema by implementing the *SchemaAdapter* interface, which allows Valence to ask the Adapter about tables and fields in a system that Adapter talks to.

An Adapter's schema may vary depending on the specific external endpoint a Link is interacting with. If you were using the Remote Salesforce Adapter to talk to two different Salesforce orgs, they would have different custom fields on their standard objects, and different custom objects from each other.

2. Filters

Even though Filters are not the origin or destination of a record, they still potentially affect the shape of the record as it is processed by that Filter.

Examples:

- The Relationships Filter that comes packaged with Valence adds the Master-Detail or Lookup field and its value to the record as it goes into Salesforce so that the relationship is populated.
- A Constants Filter might create a new field on each record called `region` and set a certain value on it.

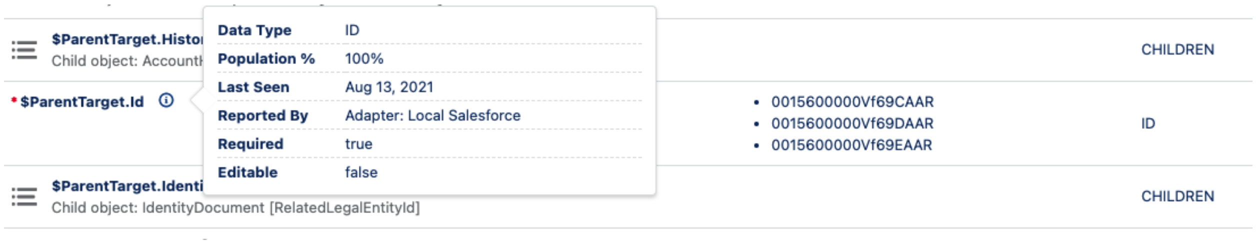
Filters can share information about their impact on the schema of a record by implementing the *SchemaAwareTransformationFilter* interface. Valence will incorporate these details into the Transformations screen so that admins can understand exactly what effect a particular Filter is having on records.

3. Link Splits

Link Splits connect two Links together, passing records from one to the other. This means that the schema from the first Link will spill over into the schema on the second Link. This concept can be a little mind-bending at first, so here's a real-world example:

> Let's say you have an "Invoice" record arriving in Salesforce, and it holds information about the Invoice itself (total, date, paid/unpaid) and also contains several line items specifying exactly what was purchased. You want to create a record in Invoice__c, and also one record in InvoiceLineItem__c for each line item on the original record. How can you do this?
>
> You set up a Link that takes an Invoice and writes to Invoice__c, and you set up a Link Split to a second Link that takes an InvoiceLineItem and writes to InvoiceLineItem__c. You of course need to get those line items out of the first record, but maybe you want to mark each line item as paid if the Invoice itself came in as paid. And also you want to populate a Master-Detail field on InvoiceLineItem__c relating it to the Invoice__c that was just created by the first Link.
>
> This means you'll want to know each of these schemata:
>
> 1. The source side of the Invoice (to know if it was paid)
> 2. The target side of the Invoice (to know its Salesforce ID that was just created)
> 3. What each line item looks like (so we can map line item fields into InvoiceLineItem__c)
> 4. The schema for InvoiceLineItem__c (so we can pick fields to write to)

Phew! That's a lot! Luckily, we've got you covered. Valence will dynamically inspect the schema of both sides of the original Link, and bring them into the child Link under the prefixes `$ParentSource` and `$ParentTarget` respectively. Here's what that looks like in the schema browser:



$ParentTarget.Histo
Child object: Account
CHILDREN

* $ParentTarget.Id

Data Type	ID
Population %	100%
Last Seen	Aug 13, 2021
Reported By	Adapter: Local Salesforce
Required	true
Editable	false

- 0015600000Vf69CAAR
- 0015600000Vf69DAAR
- 0015600000Vf69EAAR

ID

$ParentTarget.Identi
Child object: IdentityDocument [RelatedLegalEntityId]
CHILDREN

So now in your second Link (the one that writes to InvoiceLineItem__c), you can have a mapping from `$ParentTarget.Id` to the Master-Detail field `InvoiceLineItem__c.ParentInvoice__c` so that each line item stays connected to its newly-create Invoice__c record.

4. Field Custom Metadata Type

In addition to all the dynamic schema discovery, there is a special custom metadata type to represent fields that exist but for whatever reason aren't discoverable. Maybe you are working with an API that doesn't have any way to inspect its schema. You can load field definitions into this custom metadata type and Valence will make them part of what is shown to admins and available for mapping/transformation.

5. Sync Events

Finally, each Sync Event will keep track of the fields that actually exist on the records that it processed. The Sync Event will tell Valence things like population percentage, last seen date, and example values.

Typically the Sync Event is just enriching a field definition that was already provided to Valence by one of the other layers, but it also functions as a last line of defense. Even if none of the other layers mentioned a particular field, the Sync Event will catch it.

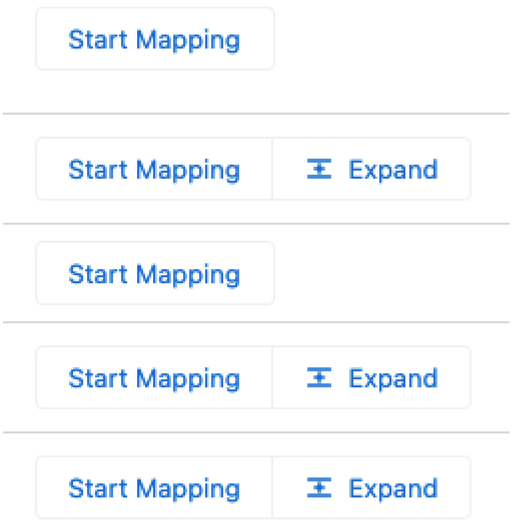
Bottom line: **If a field exists on a record, you will know it's there and as much detail about it as we can give you, and you will be able to work with it in your transformations.**

Note: The layers mentioned above are in order of importance from top to bottom. A field will be attributed to the first layer that mentions it. This means that if the only layer that saw a field is the Sync Event, in the interface you will see that it was reported by a Sync Event. However, if both an Adapter and then a Sync Event tell Valence about a field, the Sync Event is just confirming a field we were already expecting so the Adapter is considered the reporter.

Lazy Loading

Some (most?) schemata are sprawling. You are rarely working with just one entity, but also all the entities connected to it. There are relationships you could traverse until you ended up back where you started.

In order to manage all this Valence supports lazy loading for schema data. Adapters that implement the *LazyLoadSchemaAdapter* interface can be interrogated about a schema one layer at a time. As a user you will be able to do this from the schema browser screen. In the action buttons if there is an *Expand* button this field is a relationship to another entity or has nested properties that will be loaded into the screen if you click that button.



Let's say you were pulling Contact records out of Salesforce, but you also wanted to bring their parent Account's `Region__c` field. In the schema browser you would see the `Contact.Account` field as a RELATIONSHIP with an expand button. If you click on it all the fields that are on Account will load onto the screen, allowing you to see and work with `Account.Region__c`. By using `Account.Region__c` as a source field for a mapping on this Contact Link, Valence will use a SOQL relationship query and fetch both the Contact's fields and its parent Account's `Region__c` value in a single query.

5.10 Multidimensional Data

5.10.1 What Is It?

When records have complex (non-scalar) data inside one of their fields, you're dealing with multidimensional data. You could also think of this scenario as nested or compound data.

This is best explained with an example:

Listing 1: One-dimensional

```
{
        "Name" : "Acme",
        "Type" : "Financial",
        "Website" : "acme.com"
}
```

Listing 2: Multidimensional

```
{
        "Name" : "Acme",
        "Type" : "Financial",
        "Website" : "acme.com",
        "Employees" : [
                {
                        "FirstName" : "Julie",
                        "LastName" : "Smith"
                },
                {
                        "FirstName" : "Thomas",
                        "LastName" : "Lincoln"
                }
        ]
}
```

Historically, Salesforce org create/update/delete operations only accept one-dimensional data. You work with a list of Accounts, or a list of Contacts, but not an Account with embedded Contacts. There have been small steps more recently towards multidimensional record writes with things like the REST Tree API, but it's still very uncommon and that's not likely to change.

That's all fine, one dimension is typically easier to work with, but not every system out there is as consistent as Salesforce is in this domain.

Here's a sample Invoice record from the Quickbooks Online API:

```
{
  "Invoice": {
        "TxnDate": "2014-09-19",
        "domain": "QBO",
        "PrintStatus": "NeedToPrint",
        "SalesTermRef": {
          "value": "3"
        },
        "TotalAmt": 362.07,
        "Line": [
          {
                "Description": "Rock Fountain",
                "DetailType": "SalesItemLineDetail",
                "SalesItemLineDetail": {
                  "TaxCodeRef": {
                        "value": "TAX"
                  },
                  "Qty": 1,
                  "UnitPrice": 275,
                  "ItemRef": {
                        "name": "Rock Fountain",
                        "value": "5"
                  }
                },
                "LineNum": 1,
                "Amount": 275.0,
                "Id": "1"
          },
          {
                "Description": "Fountain Pump",
                "DetailType": "SalesItemLineDetail",
                "SalesItemLineDetail": {
                  "TaxCodeRef": {
                        "value": "TAX"
                  },
                  "Qty": 1,
                  "UnitPrice": 12.75,
                  "ItemRef": {
                        "name": "Pump",
                        "value": "11"
                  }
                },
                "LineNum": 2,
                "Amount": 12.75,
                "Id": "2"
          },
          {
                "Description": "Concrete for fountain installation",
                "DetailType": "SalesItemLineDetail",
                "SalesItemLineDetail": {
                  "TaxCodeRef": {
                        "value": "TAX"
                  },
```

(continues on next page)

(continued from previous page)

```
                  "Qty": 5,
                  "UnitPrice": 9.5,
                  "ItemRef": {
                        "name": "Concrete",
                        "value": "3"
                  }
                },
                "LineNum": 3,
                "Amount": 47.5,
                "Id": "3"
          },
          {
                "DetailType": "SubTotalLineDetail",
                "Amount": 335.25,
                "SubTotalLineDetail": {}
          }
        ],
        "DueDate": "2014-10-19",
        "ApplyTaxAfterDiscount": false,
        "DocNumber": "1037",
        "sparse": false,
        "CustomerMemo": {
          "value": "Thank you for your business and have a great day!"
        },
        "Deposit": 0,
        "Balance": 362.07,
        "CustomerRef": {
          "name": "Sonnenschein Family Store",
          "value": "24"
        },
        "TxnTaxDetail": {
          "TxnTaxCodeRef": {
                "value": "2"
          },
          "TotalTax": 26.82,
          "TaxLine": [
                {
                  "DetailType": "TaxLineDetail",
                  "Amount": 26.82,
                  "TaxLineDetail": {
                        "NetAmountTaxable": 335.25,
                        "TaxPercent": 8,
                        "TaxRateRef": {
                          "value": "3"
                        },
                        "PercentBased": true
                  }
                }
          ]
        },
        "SyncToken": "0",
        "LinkedTxn": [
```

(continues on next page)

(continued from previous page)

```
          {
                "TxnId": "100",
                "TxnType": "Estimate"
          }
        ],
        "BillEmail": {
          "Address": "Familiystore@intuit.com"
        },
        "ShipAddr": {
          "City": "Middlefield",
          "Line1": "5647 Cypress Hill Ave.",
          "PostalCode": "94303",
          "Lat": "37.4238562",
          "Long": "-122.1141681",
          "CountrySubDivisionCode": "CA",
          "Id": "25"
        },
        "EmailStatus": "NotSet",
        "BillAddr": {
          "Line4": "Middlefield, CA  94303",
          "Line3": "5647 Cypress Hill Ave.",
          "Line2": "Sonnenschein Family Store",
          "Line1": "Russ Sonnenschein",
          "Long": "-122.1141681",
          "Lat": "37.4238562",
          "Id": "95"
        },
        "MetaData": {
          "CreateTime": "2014-09-19T13:16:17-07:00",
          "LastUpdatedTime": "2014-09-19T13:16:17-07:00"
        },
        "CustomField": [
          {
                "DefinitionId": "1",
                "StringValue": "102",
                "Type": "StringType",
                "Name": "Crew #"
          }
        ],
        "Id": "130"
  },
  "time": "2015-07-24T10:48:27.082-07:00"
}
```

Yikes! Not only are there nested lists of records (arrays), there are directly-nested records as well (maps). So how do we get something like this to play nicely with Salesforce?

5.10.2 Working With Multidimensional Data in Valence

The best way to understand how Valence handles multidimensional is to talk about outbound *from* Salesforce and inbound *to* Salesforce separately.

Then we'll get into the nitty-gritty details of how the actual field transformations occur.

Outbound From Salesforce

If we need to retrieve data from multiple Salesforce objects to combine and send to an external system, we are going to rely on SOQL relationship queries, which we already talked about a bit in the *lazy loading* section of the *Schema* page.

We might end up pulling some scalar values in from a related record using a **child-to-parent** query like this:

```
SELECT FirstName, LastName, Account.Name, Account.Website FROM Contact
```

Which would traverse the the relationship from a Contact to its parent Account and pull down some Account fields to use in Valence mappings. The record returned by this query would look like this:

```
[
        {
                "FirstName" : "Jim",
                "LastName" : "Jonston",
                "Account" : {
                        "Name" : "Acme",
                        "Website" : "acme.com"
                }
        }
]
```

Note: Salesforce limits child-to-parent queries to *five* levels.

Alternatively (or at the same time), we can do a **parent-to-child** relationship query:

```
SELECT Name, Website, (SELECT FirstName, LastName FROM Contacts) FROM Account
```

This will fetch an Account and also nest all its related Contacts in a field called `Contacts`:

```
[
        {
                "Name" : "Acme",
                "Website" : "acme.com",
                "Contacts" : [
                        {
                                "FirstName" : "Jim",
                                "LastName" : "Jonston"
                        },
                        {
                                "FirstName" : "Samantha",
                                "LastName" : "McCay"
                        }
                ]
        }
]
```

Note: Salesforce limits parent-to-child queries to *one* level.

Valence will automatically build these queries as-needed depending on which source fields you have selected for mappings. To bring in related records, simply expand those relationships in the schema browser and start using them in mappings.

Since the target system is expecting multidimensional data, all that remains is to map to those fields and let Valence do the heavy lifting. To better understand how that actually happens, head down the page to *Transformation Mechanics*.

Inbound To Salesforce

Having multidimensional data coming into Salesforce is definitely the trickier of the two directions. Salesforce will not accept records nested inside other records, and each Link writes to only one target object. So how do we solve this dilemma?

With *Link Splits*!

You can read in much more detail about **Link Splits** at the link above, but in a nutshell a Link Split allows us to take some of the data coming into a Link and "split" it off to another Link.

So let's say we had that same record shape from just above:

```
[
        {
                "Name" : "Acme",
                "Website" : "acme.com",
                "Contacts" : [
                        {
                                "FirstName" : "Jim",
                                "LastName" : "Jonston"
                        },
                        {
                                "FirstName" : "Samantha",
                                "LastName" : "McCay"
                        }
                ]
        }
]
```

Except now we are bring this record into Salesforce. We could tackle that by setting up the following:

1. A Link whose source looks like the Acme record above and whose target is the Account object in Salesforce
2. A Link with no source and whose target is the Contact table
3. A Link Split connecting Link #1 to Link #2 and specifying the `Contacts` field as its "inner list" (again, read the Link Splits page to learn more)

Why doesn't the second Link have a source, you ask? Because it is being fed records directly from Link #1, and doesn't go out to retrieve any data of its own.

When the Link Split passes the Account record down to Link #2, because we selected an "inner list" it will actually *invert* the records and they'll look like this:

```
[
        {
```

(continues on next page)

(continued from previous page)

```
                "FirstName" : "Jim",
                "LastName" : "Jonston",
                "$ParentSource" : {
                        "Name" : "Acme",
                        "Website" : "acme.com"
                },
                "$ParentTarget" : {
                        "Id" : "0015600000Vf9A9AAJ"
                }
        },
        {
                "FirstName" : "Samantha",
                "LastName" : "McCay",
                "$ParentSource" : {
                        "Name" : "Acme",
                        "Website" : "acme.com"
                },
                "$ParentTarget" : {
                        "Id" : "0015600000Vf9A9AAJ"
                }
        }
]
```

Each record destined to become a Contact in Salesforce has its field values and also a copy of the values from its parent Account (so for example you could easily write to `Contact.AccountId` and relate the new Contact from Link #2 to the new Account created by Link #1).

If for some crazy reason you had even more nested layers of records, you can keep setting up Link Splits and new Links to peel more and more layers off this onion. You can even refer back to any parent field from any layer! Get ready to map `$ParentSource.$ParentSource.$ParentSource.Id`!

Tip: Link Splits aren't just for solving inbound multidimensional data problems. You can use them in a variety of interesting ways, like routing records to various secondary Links based some field on each record designating a category.

Transformation Mechanics

Ok, so we've learned about the broad aspects of tackling multidimensional data outbound from Salesforce and inbound to Salesforce, but how do the actual fields get mapped? Let's dig into that.

Valence works really hard to make your life easier and tricky multidimensional mappings are no exception! If Valence can figure out what you're trying to do with a mapping it will help you accomplish it. You can take a nested source field and write it to a top-level target field, no problem. You can take a top-level source field and write it to a nested target field! Valence will automatically create each layer between the target top level and the nested field you are writing to when it builds that record for delivery.

What we're really talking about here is the **Mappings Filter**, which is a special Filter that comes with Valence and is responsible for turning source fields into target fields. Below are all the permutations of how this Filter handles Mappings based on the data types of the Mapping's source and target fields.

You'll probably fall asleep trying to read through all the possible variations, so suffice it to say that as a rule of thumb just map whatever you want to whatever you want, and Valence will more than likely line things up in a way that is intuitive and makes sense for what you would have wanted to happen, without much effort on your part.

If you really are going to read through all this, here's a sample source record that will be used in each of the examples below.

Listing 3: Source Record Shape

```
{
        "preferences" : {
                "foodItem" : "ice cream",
                "flavors" : ["chocolate", "double chocolate", "mega chocolate"]
        },
        "currentAddress" : {
                "city" : "Honolulu",
                "state" : "HI"
        },
        "previousAddresses" : [
                {
                        "city" : "Boston",
                        "state" : "MA"
                },
                {
                        "city" : "New York",
                        "state" : "NY"
                }
        ]
}
```

Non-Array to Non-Array

Source: A `scalar or map` value that might or might not be nested, and is not inside any `arrays`

Target: A `scalar or map` value that might or might not be nested, and is not inside any `arrays`

The data value will be extracted from the source field (even if nested), and written to the target field. If the target field is nested, intermediary layers will be created.

Examples

Mapping: **preferences.foodItem** to **favoriteFood**

Listing 4: Target Result

```
{
        "favoriteFood" : "ice cream"
}
```

Mapping: **preferences.foodItem** to **favorites.favoriteFood**

Listing 5: Target Result

```
{
        "favorites" : {
```

(continues on next page)

(continued from previous page)

```
                "favoriteFood" : "ice cream"
        }
}
```

Array to Array

Source: An `array` of `scalars, arrays, or maps` that might or might not be nested, but is not inside any `arrays`

Target: An `array` of `scalars, arrays, or maps` that might or might not be nested, but is not inside any `arrays`

Each item in the source array will be written unchanged and in its entirety as an item in the target array. If the target field is nested, intermediary layers will be created.

Examples

Mapping: **previousAddresses** to **addresses** (where addresses is an array)

Listing 6: Target Result

```
{
        "addresses" : [
                {
                        "city" : "Boston",
                        "state" : "MA"
                },
                {
                        "city" : "New York",
                        "state" : "NY"
                }
        ]
}
```

Mapping: **previousAddresses** to **housingDetails.addresses**

Listing 7: Target Result

```
{
        "housingDetails" : {
                "addresses" : [
                        {
                                "city" : "Boston",
                                "state" : "MA"
                        },
                        {
                                "city" : "New York",
                                "state" : "NY"
                        }
                ]
        }
}
```

Array to Non-Array

Source: An `array` of `scalars, arrays, or maps` that might or might not be nested, but is not inside any `arrays`

Target: A `scalar` value that might or might not be nested, but is not inside any `arrays`

The items in the source array will be individually "stringified", joined together with commas, and written as a String to the target field. If the target field is nested, intermediary layers will be created.

Examples

Mapping: **previousAddresses** to **housingInfo**

Listing 8: Target Result

```
{
        "housingInfo" : "{city=Boston, state=MA},{city=New York, state=NY}"
}
```

Non-Array to Array

Source: A `scalar or map` value that might or might not be nested, but is not inside any `arrays`

Target: An `array` of `scalars, arrays, or maps` that might or might not be nested, but is not inside any `arrays`

The data value will be extracted from the source field (even if nested), and written unchanged and in its entirety as an item in the target array. If the target field is nested, intermediary layers will be created.

If multiple mappings are like this and write to the same array target field, each value will end up as its own item in that one array.

Examples

Mapping: **preferences.foodItem** to **personalDetails.foods** (where foods is an array)

Listing 9: Target Result

```
{
        "personalDetails" : {
                "foods" : ["ice cream"]
        }
}
```

Mapping: **currentAddress** to **personalDetails.addresses**

Listing 10: Target Result

```
{
        "personalDetails" : {
                "addresses" : [
                        {
                                "city" : "Honolulu",
                                "state" : "HI"
```

(continues on next page)

(continued from previous page)

```
                                }
                        ]
                }
        }
```

Non-Array (inside an Array) to Array

Source: A `scalar or map` value that might or might not be nested, and has an `array` somewhere above it in the source schema

Target: An `array` of `scalars, arrays, or maps` that might or might not be nested, but is not inside any `arrays`

The data value will be extracted from the source field (even if nested) **inside each item** of the ancestor array, creating an array of source data values which will become items in the target array. If the target field is nested, intermediary layers will be created.

Examples

Mapping: **previousAddresses.city** to **personalDetails.citiesVisited** (where citiesVisited is an array field)

Listing 11: Target Result

```
{
        "personalDetails" : {
                "citiesVisited" : ["Boston", "New York"]
        }
}
```

Non-Array (inside an Array) to Non-Array

Source: A `scalar or map` value that might or might not be nested, and has an `array` somewhere above it in the source schema

Target: A `scalar or map` value that might or might not be nested, but is not inside any `arrays`

The data value will be extracted from the source field (even if nested) **inside the first item** of the ancestor array, and that single value will be written to the target field. If the target field is nested, intermediary layers will be created.

Examples

Mapping: **previousAddresses.city** to **personalDetails.aPreviousCity**

Listing 12: Target Result

```
{
        "personalDetails" : {
                "aPreviousCity" : "Boston"
        }
}
```

Non-Array (inside an Array) to Non-Array (inside an Array)

Source: A `scalar or map` value that might or might not be nested, and has an `array` somewhere above it in the source schema

Target: A `scalar or map` value that might or might not be nested, and has an `array` somewhere above it in the source schema

The data value will be extracted from the source field (even if nested) **inside each item** of the ancestor array, and these data values will be written to different items inside the ancestor array of the target field. If the target field is nested, intermediary layers will be created.

Examples

Mapping: **previousAddresses.city** to **addresses.location** (where addresses is an array)

Listing 13: Target Result

```
{
        "addresses" : [
                {
                        "location" : "Boston"
                },
                {
                        "location" : "New York"
                }
        ]
}
```

Non-Array to Non-Array (inside an Array)

Source: A `scalar or map` value that might or might not be nested, but is not inside any `arrays`

Target: A `scalar or map` value that might or might not be nested, and has an `array` somewhere above it in the source schema

The data value will be extracted from the source field (even if nested), and will be written to the first item inside the ancestor array of the target field. If the target field is nested, intermediary layers will be created.

Examples

Mapping: **preferences.foodItem** to **favorites.name** (where favorites is an array)

Listing 14: Target Result

```
{
        "favorites" : [
                {
                        "name" : "ice cream"
                }
        ]
}
```

Mapping: **currentAddress** to **personalDetails.location** (where personalDetails is an array)

Listing 15: Target Result

```
{
        "personalDetails" : [
                {
                        "location" : {
                                "city" : "Honolulu",
                                "state" : "HI"
                        }
                }
        ]
}
```

5.11 Link Splits

Tip: If you haven't already looked at *Multidimensional Data*, familiarizing yourself with the ideas presented there will go a long way towards helping understand Link Splits.

5.11.1 What Is A Link Split?

Link Splits allow you to send a record to an additional Link.

The concept is simple, but the variety of new behaviors this opens up can take time to explore and understand. Let's dig in.

5.11.2 What Problems Do Link Splits Solve?

Splitting One Row of Data Into Multiple Records

Sometimes you are given a single record but you would like some fields to be placed in one table, and other fields in a different table. Perhaps you have something like this:

```
{
        "FirstName" : "Tim",
        "LastName" : "Anchor",
        "Company" : "Acme"
}
```

You'd like to take this record and turn it into an Account called "Acme" with a Contact beneath it called "Tim Anchor".

Here's how a Link Split will help. You would configure:

1. A Link writing to the Account object in Salesforce that has a mapping for `Company` -> `Account.Name`
2. A Link with no source and whose target is the Contact table with mappings for `FirstName` and `LastName`, as well as for `$ParentTarget.Id` -> `Contact.AccountId`
3. A Link Split connecting Link #1 to Link #2

Here's the flow: a record arrives at Link #1 and cherry-picks the `Company` field to make a new Account. Then the same record with all the same fields is delivered to Link #2. This Link doesn't care about the `Company` field, and instead is going to grab `FirstName` and `LastName` so it can create a new Contact. But there's a wrinkle…we want our new Contact to be related up to our new Account!

That's where `$ParentTarget` comes into play. Anytime a Link Split is used to pass a record into a new Link, that new Link has access to the FULL record from the previous Link, including what it looked like after transformations! So the second Link can work with the original version of the record, the post-transformation version of the record, or some mix of the two (like we have here). Mapping `$ParentTarget.Id` to `Contact.AccountId` will populate that Master-Detail field the way we want it, and we're happy with the result.

If you'd like to read more about `$ParentTarget` (and `$ParentSource`!), have a look at *Schema*.

Tip: If you were wondering why Link #2 has no source defined, it's because any Link run started by a Link Split doesn't need a source Adapter…we already have the records in the right format for Valence to work with from the previous Link! Sometimes you do want a data source on the second Link. Maybe you have a Link that fetches it's own records, but occasionally you also send it records from another Link. That's totally fine, and in fact a very powerful pattern that people actually use (scroll down to *Routing Records To An Appropriate Handler* to see this in action).

Breaking Down Multidimensional Data

As discussed in *Multidimensional Data*, a Link Split is a very handy way to take a record that has layers of information in it we care about, and peel one layer off at a time. This is especially important when writing records into Salesforce, which is unable to ingest and work with nested record data.

Let's look at the example from that page again:

```
[
        {
                "Name" : "Acme",
                "Website" : "acme.com",
                "Contacts" : [
                        {
                                "FirstName" : "Jim",
                                "LastName" : "Jonston"
                        },
                        {
                                "FirstName" : "Samantha",
                                "LastName" : "McCay"
                        }
                ]
        }
]
```

We want to bring this record into Salesforce and use a Link Split to process the information about the company and the information about contacts as separate efforts.

1. A Link whose source looks like the Acme record above and whose target is the Account object in Salesforce
2. A Link with no source and whose target is the Contact table
3. A Link Split connecting Link #1 to Link #2 and specifying the `Contacts` field as its "inner list"

What is an **inner list**?

This is a configuration option on the Link Split itself when you are setting it up. You can pick a single field that is an array as your inner list. What you are doing is telling Valence that you think for the next Link *the records inside this field* are really the primary records you're interested in working with. All of the outer stuff is essentially metadata from the perspective of these inner records, just some extra information that may or may not be useful.

Valence will do some special magic here if you've selected an inner list with your Link Split.

When the Link Split passes the Account record down to Link #2, Valence will actually *invert* the records and they'll look like this:

```
[
        {
                "FirstName" : "Jim",
                "LastName" : "Jonston",
                "$ParentSource" : {
                        "Name" : "Acme",
                        "Website" : "acme.com"
                },
                "$ParentTarget" : {
                        "Id" : "0015600000Vf9A9AAJ"
                }
        },
        {
                "FirstName" : "Samantha",
                "LastName" : "McCay",
                "$ParentSource" : {
                        "Name" : "Acme",
                        "Website" : "acme.com"
                },
                "$ParentTarget" : {
                        "Id" : "0015600000Vf9A9AAJ"
                }
        }
]
```

This allows you to quite easily work with these inner records in the next Link, and it makes it *especially* nice if that Link happened to already exist and already had been configured and set up with mappings and transformations that expected a certain record shape!

This means you could have a setup where you have a Link that knows how to turn some Person record shape into a Contact, and was consuming Person records and turning them into Contacts. Then you could have all kinds of Links that process records that have embedded Person data, and have splits off those Links sending all that person info to the centralized Link that knows what to do with it.

Each record sent from the Link Split to the next Link has its field values and also a copy of the values from its original record, as shown above. If you'd like to better understand `$ParentSource` and `$ParentTarget`, have a look at *Schema*.

Note: There is no limit on the number of Link Splits that can be attached to a single Link, and there is no limit on the number of times the same record can be passed down a chain of Links and Link Splits.

Routing Records To An Appropriate Handler

Picture an external system that has tables for companies, people, and deals. You can call into each table to fetch all the records in it, but there's no mechanism on the table itself to only fetch recent records. Instead, there is a special endpoint called "changesSince" and you pass it a timestamp to get all the records that have been modified since that timestamp.

Here's the problem: since it's every record that's been modified, you'll get back a mixed collection of companies, people, and deals all jumbled together. Not very convenient!

This is one way you could configure Valence to work with this external system:

1. Link that pulls from the Companies table and writes to Accounts (understands how to transform a Company record into an Account, with all the mappings and transformations that entails)
2. Link that pulls from the People table and writes to Contact (understands how to transform a Person record into a Contact, with all the mappings and transformations that entails)
3. Link that pulls from the Deals table and writes to Opportunity (understands how to transform a Deal record into an Opportunity, with all the mappings and transformations that entails)
4. Link that pulls from the changesSince endpoint but that does not have a target configured, just a data source
5. Link Split from #4 to #1, filtering out any record that isn't a Company
6. Link Split from #4 to #2, filtering out any record that isn't a Person
7. Link Split from #4 to #3, filtering out any record that isn't a Deal

Here are some takeaways from this setup:

- A Link Split can be configured to select which records are delivered to the recipient Link; you do this with a special Filter that implements the *LinkSplitFilter* interface.
- A Link doesn't have to have a target configured; the router Link doesn't really make sense to deliver records directly, its job is to send each record to the right secondary Link.
- It can be a good idea to have secondary Links still have their own data sources and do their own things with them.

Let's talk about that third point a little more. Even though in this scenario you have access to a stream of recent changes, you might still need to be able to fetch the entire table of records. Certainly for your initial data load, but perhaps also at some interval as a way to reconcile the data and make sure everything exists in both systems. With this kind of design you'd think of the data stream router as more of a supplement or enhancement to the other Links, rather than the *only* way you get records from your external system. Maybe you schedule your three main Links to fetch the entire table once a week on the weekend, or biweekly, or once a month. Whatever makes sense for your business!

Sending Data to an Additional System

Sometimes you just want to send the same record to two places. Perhaps you want to send a copy of every record that comes into Salesforce over to your data warehouse to make it available for analytics.

Set up a Link Split, and configure the second Link to push out to the data warehouse external system.

5.12 Sending Realtime Records To Salesforce

This guide will walk you through how to send records to Valence using the Salesforce APIs, which will allow you to send record events into Salesforce as they happen.

Links are surfaced using the Salesforce Apex REST API. You'll set up your Salesforce authentication and then start sending messages to the Link using an Apex REST endpoint.

5.12.1 Authentication

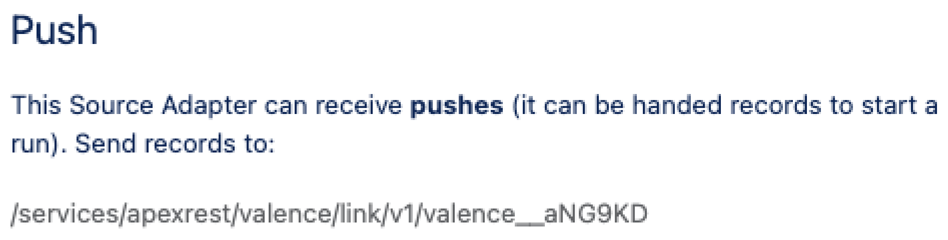
Salesforce offers a variety of ways to authenticate to its REST API. There is no special authentication for Valence; just authenticate to the Salesforce stack using standard mechanisms, which will also allow you to access the Valence endpoint.

Read: https://developer.salesforce.com/docs/atlas.en-us.api_rest.meta/api_rest/intro_understanding_authentication.htm

Tip: Unless you are using a pre-existing application it is likely you will need to set up a Connected App in Salesforce. Instructions for that are part of the link above.

5.12.2 Sending The Message

Once authenticated, sending records to Valence is straightforward. Each Link has a unique endpoint URL that can be used to send records to kick off a Link run. You can find that URL in the top left corner of the Link dashboard:



Push

This Source Adapter can receive **pushes** (it can be handed records to start a run). Send records to:

/services/apexrest/valence/link/v1/valence__aNG9KD

The base URL is given to you when you authenticate with Salesforce. One of the parameters returned during authentication is your "instance url", which will look something like this:

> https://connect-innovation-3618-dev-ed.cs62.my.salesforce.com

So in this case, my full URL for the Link would be:

> https://connect-innovation-3618-dev-ed.cs62.my.salesforce.com/services/apexrest/valence/link/v1/valence__aAtnnc

To submit records, send an HTTPS **POST** to this endpoint (and don't forget to include your authentication information in the header!).

The body of your post will be passed to the Link's *Source Adapter* for processing. The expected shape of that body is specific to each source Adapter and how it was coded (so check the documentation for the Adapter you're using). As a best practice we recommend a JSON body format that looks something like this:

Possible Message Shape

```
[
    {
        "Name" : "First Record",
        "Favorite Color" : "Blue"
    },
    {
        "Name" : "Second Record",
        "Favorite Color" : "You Get the idea..."
    }
]
```

5.13 Evaluating an API for Ease-of-Use With Valence

Almost always, if an API is addressable via HTTP or HTTPS, you can use Valence to talk to it.

Now, how *easy* that will be greatly depends on how the API is constructed, and what sort of features it supports. This is a companion guide to *How to Design a Complementary Valence API*. That guide talks about designing an API, this guide is for evaluating a preexisting API and gives you a checklist to go through to see what level of effort it would take to talk to it from Valence.

This is a guide for someone designing/developing a new Valence Adapter. If an Adapter already exists for the API you need to connect to, you can stop reading. You're all set. If not, here are the things you should investigate:

5.13.1 Topics of Investigation

Authentication

How does this API handle authentication? Is it an API key? An OAuth flow? Salesforce handles some basic scenarios out of the box with NamedCredentials. If the authentication for the API is standard then there's almost no effort here. If it's something custom you will likely need to write code to retrieve tokens or keys to use.

Schema

How will we know about the schema of the API?

Best case scenario there is an endpoint (or endpoints) in the API itself that self-describe the API. For example, a common setup is an endpoint to list tables, and another endpoint to list fields, given a table name. Dynamic schema discovery is a huge win, as it will allow Valence to surface those details to users, and stay current as the API changes over time.

A next-best-option that sometimes exists is a static file that represents the schema. Perhaps this is a Swagger or Open API document, or some kind of JSON / XML representation of the schema (like a WSDL). If this is the case, it can be a good pattern to store that file as a static resource in Salesforce, and load it from your Apex code to interpret at runtime when doing schema work for the user.

A last resort is to either hardcode the schema into the Apex class, or give up completely and leave the schema unknown. An unknown schema means the user will have nothing to map against, but if records flow Valence will discover the record shape and surface it to users. Not nearly as good as the other options, but can help in a pinch.

5.13.2 Research Checklist

Use this list of questions to gather relevant details to drive your design:

Authentication

1. What authentication options exist? Basic auth? API key? OAuth? If OAuth, what flows are supported?

Schema

2. Is the schema retrievable via API? What does that look like?
3. If not retrievable via API, is the schema stored in a machine-readable format somewhere? WSDL / Swagger / Open API / etc?
4. If not machine-readable, how is the schema documented?

Reading

5. Is it possible to fetch records from the API?
6. In what format are records received (CSV, JSON, XML, etc)?
7. Can specific columns/fields be selected, or is always full records?
8. Can records be filtered in some way, ideally based on a last-modified timestamp or similar?
9. Is record fetching synchronous or asynchronous?
10. How many records can be retrieved at once?
11. Are read results paginated? If so, what is the mechanism?
12. If pagination, can all the pages be calculated upfront (ex: knowing a total count of records) or do we learn along the way (ex: there is a next page, but we don't know how many more next pages there will be)?

Writing

13. Is it possible to write records to the API?
14. In what format are records written (CSV, JSON, XML, etc)?
15. Do records have to be written one at a time or can they be written in bulk? How many records can be written at once?
16. Is upsert a supported operation?
17. What keys/identifiers are supported for upsert/update operations?
18. Is record writing synchronous or asynchronous?
19. What does the response payload for a write operation look like? Does it include identifiers when records are created?

Other

20. What does error handling and error reporting look like?
21. Compression: Is the Accept-Encoding HTTP header supported, specifically for "gzip"?
22. Are there any sort of rate or volume limits on the API?
23. How are dates and date-times formatted? Are they UTC, or some other timezone?

5.14 How to Design a Complementary Valence API

Valence offers a lot of flexibility in how it talks to external systems, but there are best practices that make for a smoother build or that mitigate common pitfalls.

5.14.1 API Features

Must Haves

There are two important API features that will assist Valence in providing an efficient and scalable integration:

1. **Last modified date as a timestamp filter**
   Being able to pass a timestamp (date + time) value to the API allows us to get only the records that have changed since the last time records were retrieved. Two timestamps can be used to demarcate the start and stop of a range to retrieve, or a single timestamp can be used and the implication is that the end of the date range is right now. This makes the connection efficient in that we're only moving records that have changed.
2. **Paginated result sets**
   Breaking large result sets into smaller batches is a standard way to handle large data volume. This functionality is baked into Valence, for example in the two-step fetch process in the *SourceAdapterForPull* interface. The two big benefits to baking pagination into your API integration from the start are that you can move the entire data set (for an initial load or a reload) and you can handle unexpected spikes in volume. (The number of times we've seen someone dump a static file into a database and it blows up downstream integrations. . .) Most industry-standard pagination patterns will work with Valence. You can paginate on page numbers or on an offset value, either is fine. Some patterns we've seen (all fine with Valence):

   a. One endpoint to call to get a count, another to call repeatedly to get pages of records based on the result from the first endpoint (**determinate**)

   b. A single endpoint that returns metadata alongside a result, such as a total count or information about pages that can be used to fetch the remaining pages (**determinate**)

   c. A single endpoint that returns a token or value of some kind alongside a result, indicating that more records exist (but not how many) and where/how to retrieve them (**indeterminate**)

Should Haves

Valence really shines when it is able to be paired with a dynamic API that can reveal its structure as well as retrieve different data tables based on passed parameters.

1. **Self-describing schema**
   One thing an external API can offer that, together with Valence, will really empower data admins is an endpoint that describes what tables and fields are accessible. This allows an admin in Salesforce to pick and choose which tables and fields they are interested in, and change their mind down the road with minimal overhead.
2. **Dynamic fetch endpoint**
   Complementary to a self-describing schema is an endpoint that accepts the table and fields that should be queried, alongside other filters such as timestamp.
3. **Compact data encoding format**
   In order to fit as many records as possible into each page, it's best to use a space-efficient exchange format. Some examples, in order from most compact to least compact:

   a. CSV

   b. JSON

   c. XML

4. **Compression**
   An ideal API supports the Accept-Encoding header so that results can be compressed: https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/Accept-Encoding

Tip: Compressed requests coming from Salesforce to your API use `gzip` compression, and only accept `gzip` in return.

Nice to Haves

5. **Additional filters exposed in addition to timestamp**
   It's not uncommon for end users to want to filter source data on other fields, such as a type of some kind. Records can be filtered out on the Valence side after reception, but it's always nice to not transmit them to begin with.

Authentication

Valence supports these authentication methods out of the box:

1. Standard username + password header authentication
2. OAuth using the "authorization code" grant type
3. AWS Signature Version 4
4. JWT
5. JWT Token Exchange

You can still use a custom authentication mechanism or use one of the other OAuth grant types, it just requires some additional effort at the Adapter level to bake that in.

5.14.2 Example Ideal API for Fetching Data

To make these recommendations more concrete, here's a breakdown of an example API written to expose data to Valence/Salesforce that supports our recommended best practices. This API would allow an admin to browse and select whichever tables they were interested in pulling into Salesforce. The entire table would be pulled over in a series of batches/pages, and then incremental delta sync updates would keep Salesforce up-to-date.

Endpoints:

1. ***/describe-tables***
   a. **Parameters**: None
   b. **Returns**: A list of table representations (just names at a minimum, but always nice to have name, label, description, etc)
2. ***/describe-fields***
   a. **Parameters:**
      i. tableName - The name of a table that was returned by describe-tables
   b. **Returns**: A list of field representations (just names at a minimum, but always nice to have name, label, description, data type, default value, etc)
3. ***/count-records***

   a. **Parameters:**
      i. tableName - The name of a table that was returned by describe-tables
      ii. start - A timestamp for "last modified" start of range
      iii. end - A timestamp for "last modified" end of range
   b. **Returns**: A record count for how many records we would expect to receive were we to call fetch-records

4. ***/fetch-records***
   a. **Parameters:**
      i. tableName - The name of a table that was returned by describe-tables
      ii. start - A timestamp for "last modified" start of range
      iii. end - A timestamp for "last modified" end of range
      iv. fieldList - A list of field names that come from describe-fields, and is likely a subset of the total fields available (and has been selected by an admin through building mappings)
      v. offset - The number of records to skip into the total result set
      vi. pageSize - How many records to return per page
   b. **Returns**: A single page of records from the <tableName> table, selecting the <fieldList> columns, with a last modified date on each record between <start> and <end>. No more than <pageSize> records returned per page, with an offset into the total result set of <offset> records.

5.14.3 Writing To Your API

Valence supports both reading from and writing to external APIs. When writing, it is best if there is a consistent pattern for specifying which table is being written to.

It is highly appreciated when an external API supports bulk writing, i.e. writing more than one record in a single HTTP invocation.

Here is an ideal endpoint for writing to:

1. */<table_name>*

   a: **HTTP Method**: POST

   b: **Parameters**:

   i. an array of record objects (in whatever format your API is written in…JSON, CSV, XML, etc)

   c: **Returns**:

   i. Success or failure of each record individually, and/or the success/failure of the entire operation

   ii. Unique identifiers in your system for each record, especially ones that were just created because of this operation

5.15 Building a Configurable Filter that Ignores Records Based on User-selected Cutoff Date

Here we'll take a slightly more advanced scenario and walk through it.

5.15.1 Desired Behavior

- A User can apply a condition to an incoming date field where that field is evaluated and if it fails our test, the record is ignored.
- The test will be comparing the record's date in the field to a cutoff/threshold date that has been configured by the User.
- A User should be able to apply this condition to any incoming date field, and (if desired) more than one date field on the same record.
- The cutoff dates are configured per-mapping, so if multiple fields should be inspected they can have different cutoff dates.

5.15.2 Solution Walkthrough

To satisfy these expectations we'll want to create a *custom Filter* that implements both *TransformationFilter* and *ConfigurablePerMappingFilter*.

Our configuration needs are pretty simple (just a date picker), so we'll use the *Configuration Structure* pattern for handling configurations.

Class Declaration

We start out with our class declaration and implementing both of our interfaces.

```
/**
 * Valence filter that allows us to set a date threshold that will cause records to be␣
↪ignored if a given
 * field from that record is older than our threshold.
 */
global with sharing class IgnoreOldRecordsFilter implements valence.TransformationFilter,
↪ valence.ConfigurablePerMappingFilter {
```

Warning: Make sure you declare your class as **global**, otherwise Valence won't be able to see it and use it!

Configuration Setup

Since we opted for *a configuration structure*, we'll be returning **null** from getMappingConfigurationLightningComponent().

The shape we return from getMappingConfigurationStructure() will be used by Valence to build a UI on our behalf and show it to the User. We are going to use the "date" flavor of lightning:input by setting the "type" attribute on that base component so that our User gets a nice, friendly date picker.

Valence will save the User-selected date to the database for us, and we'll be given the value back later when we need it.

```
    public String getMappingConfigurationLightningComponent() {
        return null;
    }

    public String getMappingConfigurationStructure() {
                return DynamicUIConfigurationBuilder.create('Select a date below. Any␣
↪records that have a value in this mapping older than the selected date will be ignored␣
↪(exact date matches are not ignored).')
                        .addField(
                                DynamicUIConfigurationBuilder.createField('cutoff')
                                .addAttribute('label', 'Cutoff Date')
                                .addAttribute('type', 'date')
                        )
                        .finalize();
    }
```

Tip: We don't have to set "componentType" on the **cutoff** field because lightning:input is the default component type.

Configuration Explanation

We always want to give the User useful information about what they've done and what they can expect. Valence uses explainMappingConfiguration() to give us an opportunity to interpret a configuration and break it down in plain language for the User.

```
    public String explainMappingConfiguration(String configuration) {

        String explanation = 'This Filter will set records to be ignored if their field␣
↪value (the one in this mapping) is older than {0}.';

        Configuration config = (Configuration)JSON.deserialize(configuration,␣
↪IgnoreOldRecordsFilter.Configuration.class);

        return String.format(explanation, new List<String>{String.valueOf(config.cutoff)}
↪);
    }
```

Configuration Class

For convenience and cleanliness it's a good idea to create a simple inner Apex class to hold your configuration structure. Valence serializes configuration values from the form the User filled out into a JSON object whose keys are the **name** values you specified in your configuration schema. In our case we defined a single field called **cutoff** that we expect to find a serialized Date value inside.

```
    /**
     * Simple class for holding the configuration needed for this filter.
     */
    private class Configuration {
        private Date cutoff;
```

Restricting Filter Usage

Some Filters only make sense in specific scenarios, for example RelationshipFilter (the built-in Valence Filter that handles populating Master-Detail and Lookup fields) only makes sense for records flowing into Salesforce, not outbound.

For this cutoff Filter we are building, we aren't going to restrict it to only certain Links. All Links can use it.

```
    public Boolean validFor(valence.LinkContext context) {
        return true;
    }
```

Processing Records

Finally, we get into the core purpose of our Filter: ignoring old records. Let's walk through our process() method.

1. Set up a Map we will use to line up the names of the record fields we're going to inspect with the configured cutoff date for each field.

```
    public void process(valence.LinkContext context, List<valence.RecordInFlight>␣
↪records) {

        Map<String, Date> cutoffsBySourceField = new Map<String, Date>();
```

2. Iterate through the *Mapping* instances we are given as part of the *LinkContext*. Remember that Valence is clever here and will inject serialized User configurations from the database into mapping.configuration properties wherever the User has set up a configuration.
3. We collect the **cutoff** Date values from a deserialized Configuration instance for any populated configurations.

```
        for(valence.Mapping mapping : context.mappings.values()) {

            // skip blank configurations
            if(String.isNotBlank(mapping.configuration)) {
                Configuration config = (Configuration)JSON.deserialize(mapping.
↪configuration, IgnoreOldRecordsFilter.Configuration.class);
                cutoffsBySourceField.put(mapping.sourceFieldName, config.cutoff);
            }
        }
```

4. Now that we've assembled our Map, if it's empty we can stop processing.

```
        // bail out if we didn't find any
                if(cutoffsBySourceField.isEmpty()) {
                        return;
```

5. Now we iterate through the incoming *RecordInFlight* instances.

```
        for(valence.RecordInFlight record : records) {
```

6. For each field we need to check, inspect that field's value for this record.

```
            for(String sourceField : cutoffsBySourceField.keySet()) {
                Date cutoff = cutoffsBySourceField.get(sourceField);
```

7. Compare the field value to our cutoff date for this field. If older than the cutoff, mark this record as *ignored*.

```
                Long fieldValue = (Long)record.getOriginalProperties().get(sourceField);
                if(fieldValue != null) {
                    if(Datetime.newInstance(fieldValue).dateGmt() < cutoff) {
                        record.ignore('Record field <' + sourceField + '> older than␣
↪cutoff of ' + cutoff);
```

Hint: You can see in the code for this example scenario we are assuming that all dates are being transmitted as Long values, i.e. milliseconds since Epoch. This is a simplification and you may not be able to make this same assumption in your real-world scenario!

5.15.3 Full Solution Code

Here is the complete solution code that we walked through above.

```
/**
 * Valence filter that allows us to set a date threshold that will cause records to be␣
↪ignored if a given
 * field from that record is older than our threshold.
 */
global with sharing class IgnoreOldRecordsFilter implements valence.TransformationFilter,
↪ valence.ConfigurablePerMappingFilter {

    public String getMappingConfigurationLightningComponent() {
        return null;
    }

    public String getMappingConfigurationStructure() {
                return DynamicUIConfigurationBuilder.create('Select a date below. Any␣
↪records that have a value in this mapping older than the selected date will be ignored␣
↪(exact date matches are not ignored).')
                        .addField(
                                DynamicUIConfigurationBuilder.createField('cutoff')
                                .addAttribute('label', 'Cutoff Date')
                                .addAttribute('type', 'date')
                        )
                        .finalize();
```

(continues on next page)

(continued from previous page)

```
    }

    public String explainMappingConfiguration(String configuration) {

        String explanation = 'This Filter will set records to be ignored if their field value (the one in this mapping) is older than {0}.';

        Configuration config = (Configuration)JSON.deserialize(configuration, IgnoreOldRecordsFilter.Configuration.class);

        return String.format(explanation, new List<String>{String.valueOf(config.cutoff)});
    }

    public Boolean validFor(valence.LinkContext context) {
        return true;
    }

    public void process(valence.LinkContext context, List<valence.RecordInFlight> records) {

        Map<String, Date> cutoffsBySourceField = new Map<String, Date>();

        /*
         * Assemble any cutoffs that have been configured by admins.
         */
        for(valence.Mapping mapping : context.mappings.values()) {

            // skip blank configurations
            if(String.isNotBlank(mapping.configuration)) {
                Configuration config = (Configuration)JSON.deserialize(mapping.configuration, IgnoreOldRecordsFilter.Configuration.class);
                cutoffsBySourceField.put(mapping.sourceFieldName, config.cutoff);
            }
        }

        // bail out if we didn't find any
                if(cutoffsBySourceField.isEmpty()) {
                        return;
                }

        /*
         * Iterate through our records, ignoring where appropriate based on cutoff dates.
         */
        for(valence.RecordInFlight record : records) {
            for(String sourceField : cutoffsBySourceField.keySet()) {
                Date cutoff = cutoffsBySourceField.get(sourceField);
                Long fieldValue = (Long)record.getOriginalProperties().get(sourceField);
                if(fieldValue != null) {
                    if(Datetime.newInstance(fieldValue).dateGmt() < cutoff) {
                        record.ignore('Record field <' + sourceField + '> older than cutoff of ' + cutoff);
```

(continues on next page)

(continued from previous page)

```
                    }
                }
            }
        }
    }

    /**
     * Simple class for holding the configuration needed for this filter.
     */
    private class Configuration {
        private Date cutoff;
    }
}
```

5.16 Using Valence Between Two Salesforce Orgs (aka Salesforce to Salesforce)

You can use Valence to move data between two (or more) Salesforce orgs. You will only need to install Valence in one org to do this (in this guide we'll call the org that has Valence installed the "**controlling org**").

Tip: All configuration work for Valence will take place in the org where Valence is installed (the "controlling org"). Any Sync Event or Record Snapshot records for diagnostics will be saved in this org as well.

Setting up Links in Valence to move data is the easy part. The harder part is setting up your authentication between the Salesforce orgs. Let's start there.

We will need to set up three things:

1. Connected App
2. Auth. Provider
3. Named Credential

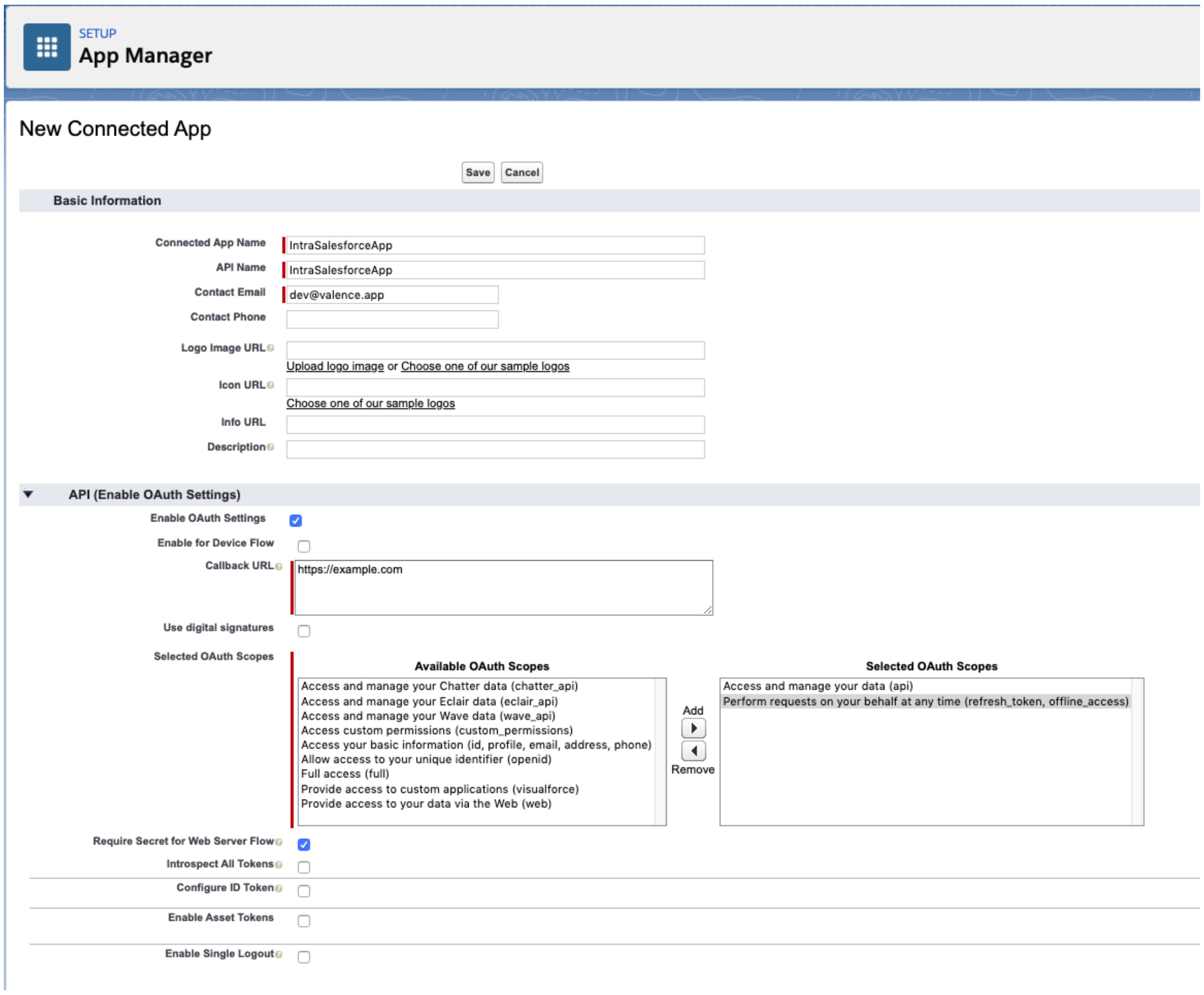
A Connected App defines an entity that will be calling **into** a Salesforce org, and an Auth Provider defines how to call **out** from a Salesforce org. Since we are communicating between two Salesforce orgs, we need both—from the perspective of the remote org it's inbound (Connected App), and from the perspective of the controlling org where Valence is installed it's outbound (Auth Provider).

You'll define **both the Connected App and the Auth Provider in the controlling org**. How is that possible, you say? Well, Salesforce is clever and propagates Connected App definitions to all orgs, so a Connected App defined in any org can be used in any other org.

If you really wanted to, you could create the Connected App inside the remote org instead. This would give you some additional configuration options, but those aren't that useful for Valence so we recommend keeping it simple and defining everything in the org where Valence is installed.

5.16.1 Set Up The Connected App

Create a new Connected App in the Setup menu of your controlling org (the one that has Valence installed). Go to Setup -> Apps -> App Manager and click "New Connected App".


SETUP
App Manager
Save
Cancel
Upload logo image or Choose one of our sample logos
Choose one of our sample logos
Add
Remove

New Connected App

Basic Information

Connected App Name	IntraSalesforceApp
API Name	IntraSalesforceApp
Contact Email	dev@valence.app
Contact Phone	
Logo Image URL	
Icon URL	
Info URL	
Description	

API (Enable OAuth Settings)

Enable OAuth Settings	✓
Enable for Device Flow	☐
Callback URL	https://example.com
Use digital signatures	☐

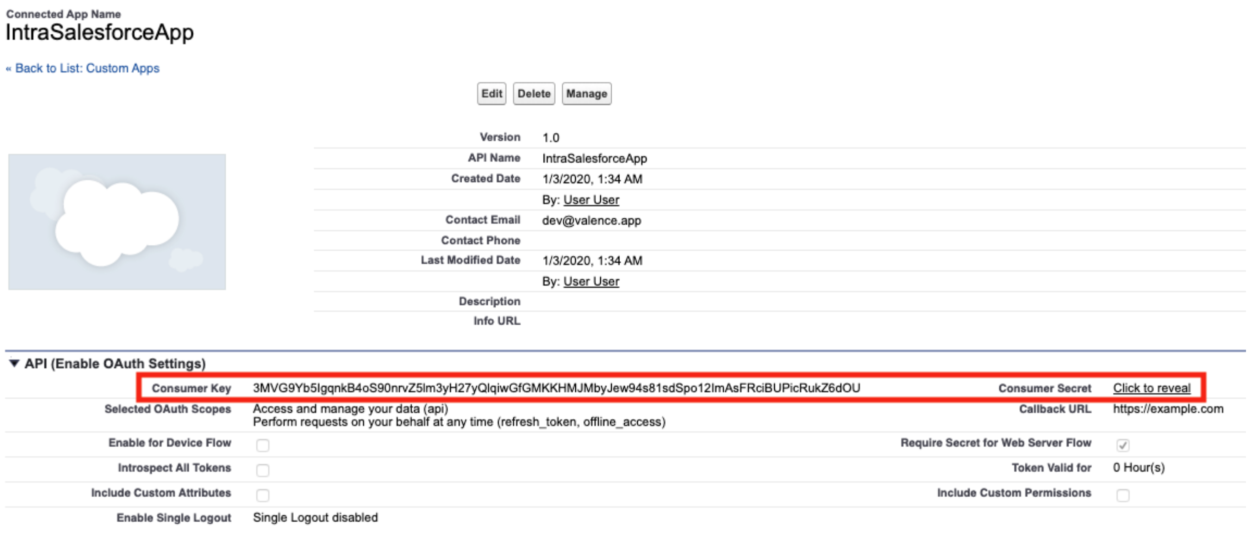
Selected OAuth Scopes

Available OAuth Scopes	Selected OAuth Scopes
Access and manage your Chatter data (chatter_api)	Access and manage your data (api)
Access and manage your Eclair data (eclair_api)	Perform requests on your behalf at any time (refresh_token, offline_access)
Access and manage your Wave data (wave_api)	
Access custom permissions (custom_permissions)	
Access your basic information (id, profile, email, address, phone)	
Allow access to your unique identifier (openid)	
Full access (full)	
Provide access to custom applications (visualforce)	
Provide access to your data via the Web (web)	

Require Secret for Web Server Flow	✓
Introspect All Tokens	☐
Configure ID Token	☐
Enable Asset Tokens	☐
Enable Single Logout	☐

1. Name it however you like and set an appropriate email address
2. Make sure "Enable OAuth Settings" is checked
3. For the Callback URL, put something temporary (like https://example.com); we'll come back to this a few steps from now
4. Select the two scopes you see in the picture (api, refresh_token)
5. Make sure "Require Secret for Web Server Flow" is checked

Once you save, you will need to copy the Consumer Key and Consumer Secret and use them in the next section.



Connected App Name

IntraSalesforceApp

« Back to List: Custom Apps

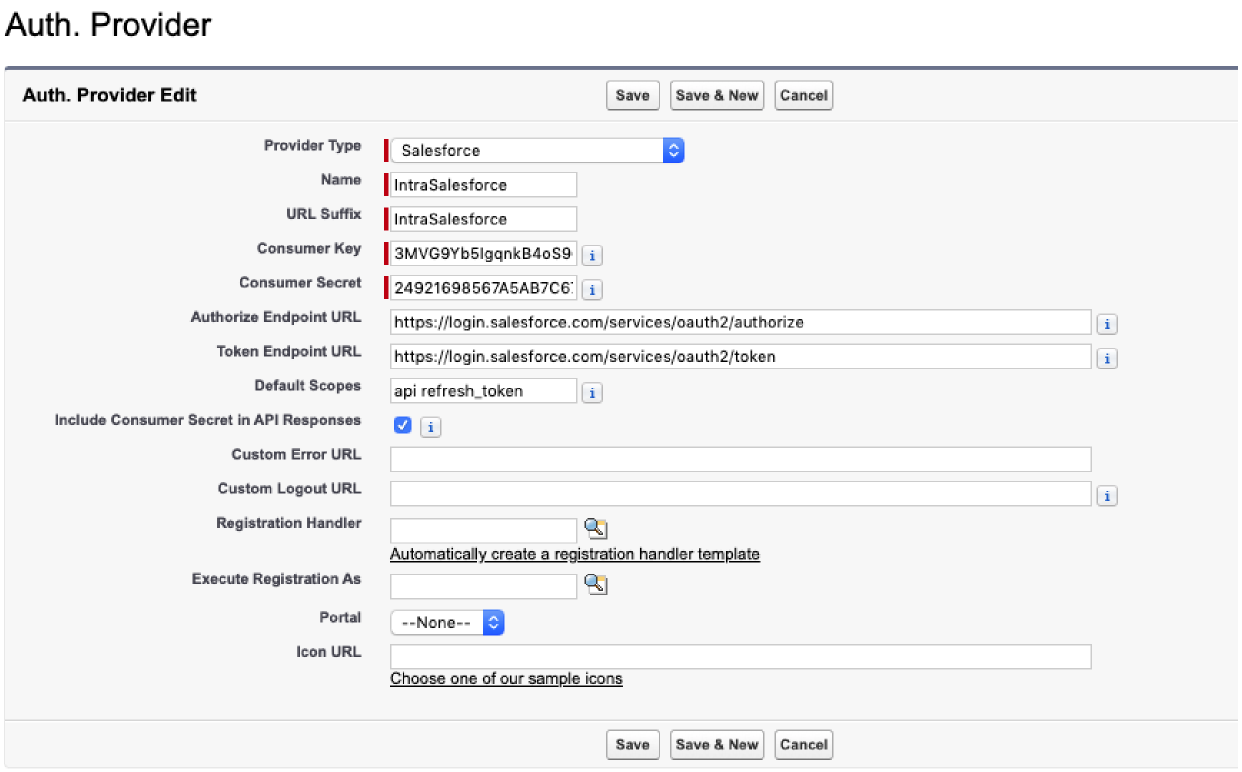
Version	1.0
API Name	IntraSalesforceApp
Created Date	1/3/2020, 1:34 AM By: User User
Contact Email	dev@valence.app
Contact Phone	
Last Modified Date	1/3/2020, 1:34 AM By: User User
Description	
Info URL	

API (Enable OAuth Settings)

Consumer Key	3MVG9Yb5IgqnkB4oS90nrvZ5lm3yH27yQlqiwGfGMKKHMJMbyJew94s81sdSpo12lmAsFRciBUPicRukZ6dOU	Consumer Secret	Click to reveal
Selected OAuth Scopes	Access and manage your data (api) Perform requests on your behalf at any time (refresh_token, offline_access)	Callback URL	https://example.com
Enable for Device Flow	☐	Require Secret for Web Server Flow	✓
Introspect All Tokens	☐	Token Valid for	0 Hour(s)
Include Custom Attributes	☐	Include Custom Permissions	☐
Enable Single Logout	Single Logout disabled		

5.16.2 Set Up The Auth Provider

Now we will create an Auth Provider record. Go to Setup -> Identity -> Auth. Providers



Auth. Provider

Auth. Provider Edit

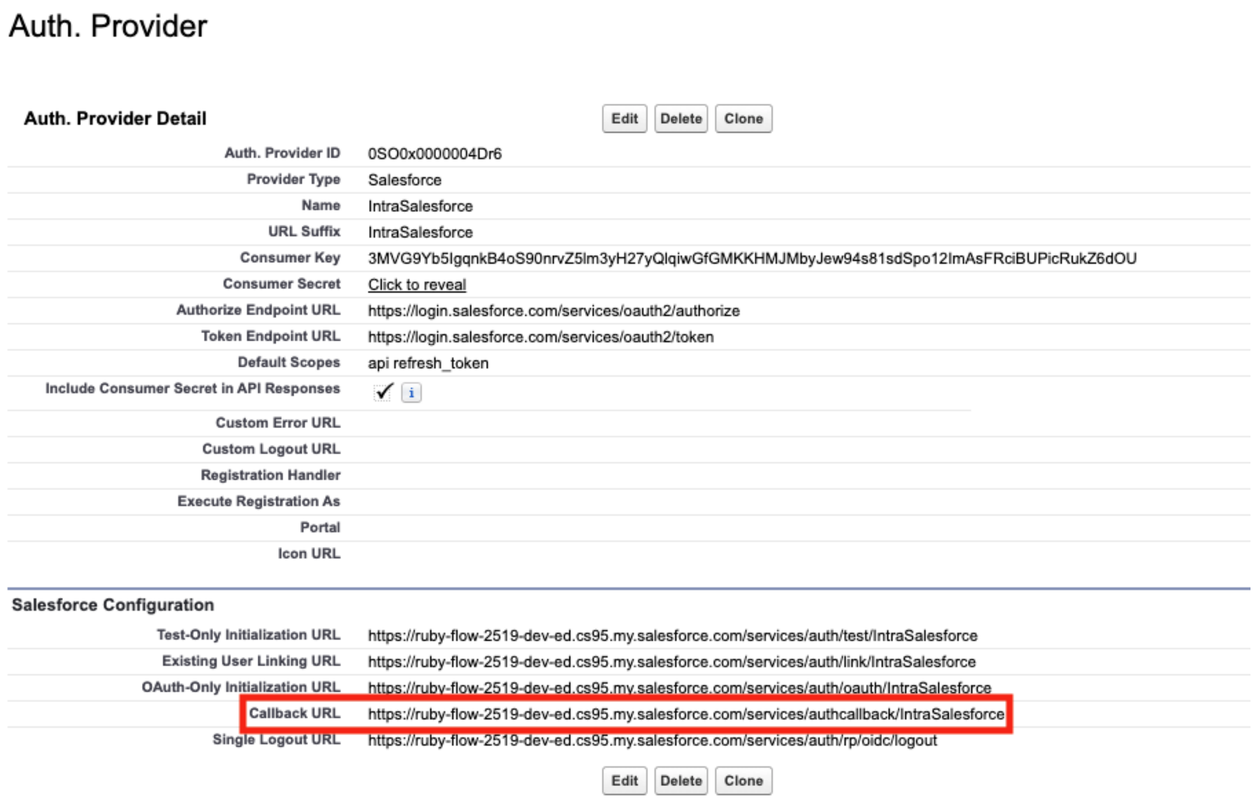
Provider Type	Salesforce
Name	IntraSalesforce
URL Suffix	IntraSalesforce
Consumer Key	3MVG9Yb5IgqnkB4oS9
Consumer Secret	24921698567A5AB7C6
Authorize Endpoint URL	https://login.salesforce.com/services/oauth2/authorize
Token Endpoint URL	https://login.salesforce.com/services/oauth2/token
Default Scopes	api refresh_token
Include Consumer Secret in API Responses	✓
Custom Error URL	
Custom Logout URL	
Registration Handler	Automatically create a registration handler template
Execute Registration As	
Portal	--None--
Icon URL	Choose one of our sample icons

1. Provider Type is "Salesforce"
2. Name is whatever you want
3. Copy and paste Consumer Key and Consumer Secret from your Connected App that you created
4. Type out the endpoints using either test.salesforce.com or login.salesforce.com (**match the remote org**

you want to connect to) https://login.salesforce.com/services/oauth2/authorize https://login.salesforce.com/services/oauth2/token

5. Set the default scopes to the same as your connected app (api refresh_token)
6. Make sure "Include Consumer Secret in API Responses" is checked

After saving the Auth Provider some URLs will be generated. Copy the Callback URL... we're about to go update our Connected App.


Auth. Provider

Auth. Provider Detail

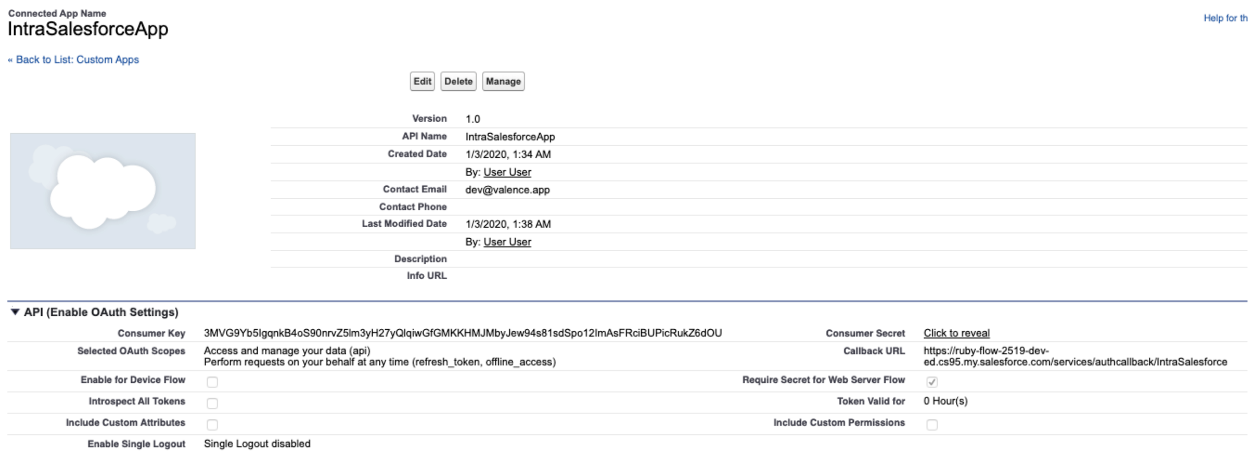
Auth. Provider ID	0SO0x0000004Dr6
Provider Type	Salesforce
Name	IntraSalesforce
URL Suffix	IntraSalesforce
Consumer Key	3MVG9Yb5IgqnkB4oS90nrvZ5lm3yH27yQlqiwGfGMKKHMJMbyJew94s81sdSpo12lmAsFRciBUPicRukZ6dOU
Consumer Secret	Click to reveal
Authorize Endpoint URL	https://login.salesforce.com/services/oauth2/authorize
Token Endpoint URL	https://login.salesforce.com/services/oauth2/token
Default Scopes	api refresh_token
Include Consumer Secret in API Responses	✓
Custom Error URL	
Custom Logout URL	
Registration Handler	
Execute Registration As	
Portal	
Icon URL	

Salesforce Configuration

Test-Only Initialization URL	https://ruby-flow-2519-dev-ed.cs95.my.salesforce.com/services/auth/test/IntraSalesforce
Existing User Linking URL	https://ruby-flow-2519-dev-ed.cs95.my.salesforce.com/services/auth/link/IntraSalesforce
OAuth-Only Initialization URL	https://ruby-flow-2519-dev-ed.cs95.my.salesforce.com/services/auth/oauth/IntraSalesforce
Callback URL	https://ruby-flow-2519-dev-ed.cs95.my.salesforce.com/services/authcallback/IntraSalesforce
Single Logout URL	https://ruby-flow-2519-dev-ed.cs95.my.salesforce.com/services/auth/rp/oidc/logout

5.16.3 Update The Connected App Callback URL

Go back and edit the Connected App, replacing the dummy Callback URL we originally gave it with the one from the Auth Provider you just created.



Connected App Name
IntraSalesforceApp

« Back to List: Custom Apps

Help for th

Version	1.0
API Name	IntraSalesforceApp
Created Date	1/3/2020, 1:34 AM
	By: User User
Contact Email	dev@valence.app
Contact Phone	
Last Modified Date	1/3/2020, 1:38 AM
	By: User User
Description	
Info URL	

▼ API (Enable OAuth Settings)

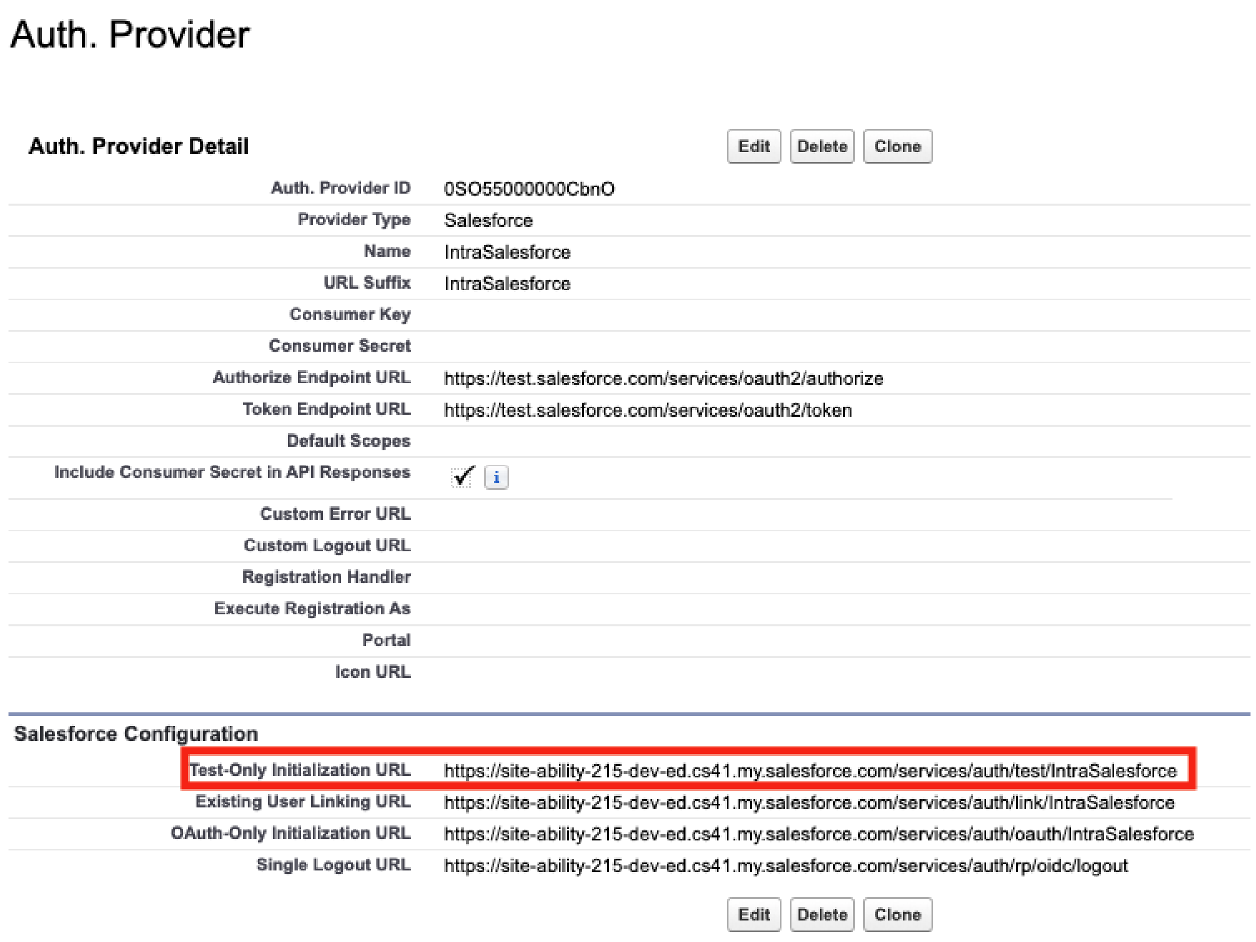
Consumer Key	3MVG9Yb5IgqnkB4oS90nrvZ5lm3yH27yQlqiwGfGMKKHMJMbyJew94s81sdSpo12lmAsFRciBUPicRukZ6dOU	Consumer Secret	Click to reveal
Selected OAuth Scopes	Access and manage your data (api) Perform requests on your behalf at any time (refresh_token, offline_access)	Callback URL	https://ruby-flow-2519-dev-ed.cs95.my.salesforce.com/services/authcallback/IntraSalesforce
Enable for Device Flow	☐	Require Secret for Web Server Flow	✓
Introspect All Tokens	☐	Token Valid for	0 Hour(s)
Include Custom Attributes	☐	Include Custom Permissions	☐
Enable Single Logout	Single Logout disabled		

It's important that the Callback URL defined in your Connected App matches the Auth Provider Callback URL. That's because the OAuth flow defines something called a "redirect_uri" that is used in more secure flows to mitigate certain attacks.

Tip: It is actually possible to define multiple Callback URLs on a Connected App (read the docs), so if you need to make multiple Auth Providers you could still use the same Connected App, if you wanted to.

5.16.4 Test Your Work So Far

Before moving on to Named Credentials, you can test that you successfully got your Connected App and Auth Provider working together by going to the Test-Only Initialization URL:



Auth. Provider

Auth. Provider Detail

Auth. Provider ID	0SO55000000CbnO
Provider Type	Salesforce
Name	IntraSalesforce
URL Suffix	IntraSalesforce
Consumer Key	
Consumer Secret	
Authorize Endpoint URL	https://test.salesforce.com/services/oauth2/authorize
Token Endpoint URL	https://test.salesforce.com/services/oauth2/token
Default Scopes	
Include Consumer Secret in API Responses	✓
Custom Error URL	
Custom Logout URL	
Registration Handler	
Execute Registration As	
Portal	
Icon URL	

Salesforce Configuration

Test-Only Initialization URL	https://site-ability-215-dev-ed.cs41.my.salesforce.com/services/auth/test/IntraSalesforce
Existing User Linking URL	https://site-ability-215-dev-ed.cs41.my.salesforce.com/services/auth/link/IntraSalesforce
OAuth-Only Initialization URL	https://site-ability-215-dev-ed.cs41.my.salesforce.com/services/auth/oauth/IntraSalesforce
Single Logout URL	https://site-ability-215-dev-ed.cs41.my.salesforce.com/services/auth/rp/oidc/logout

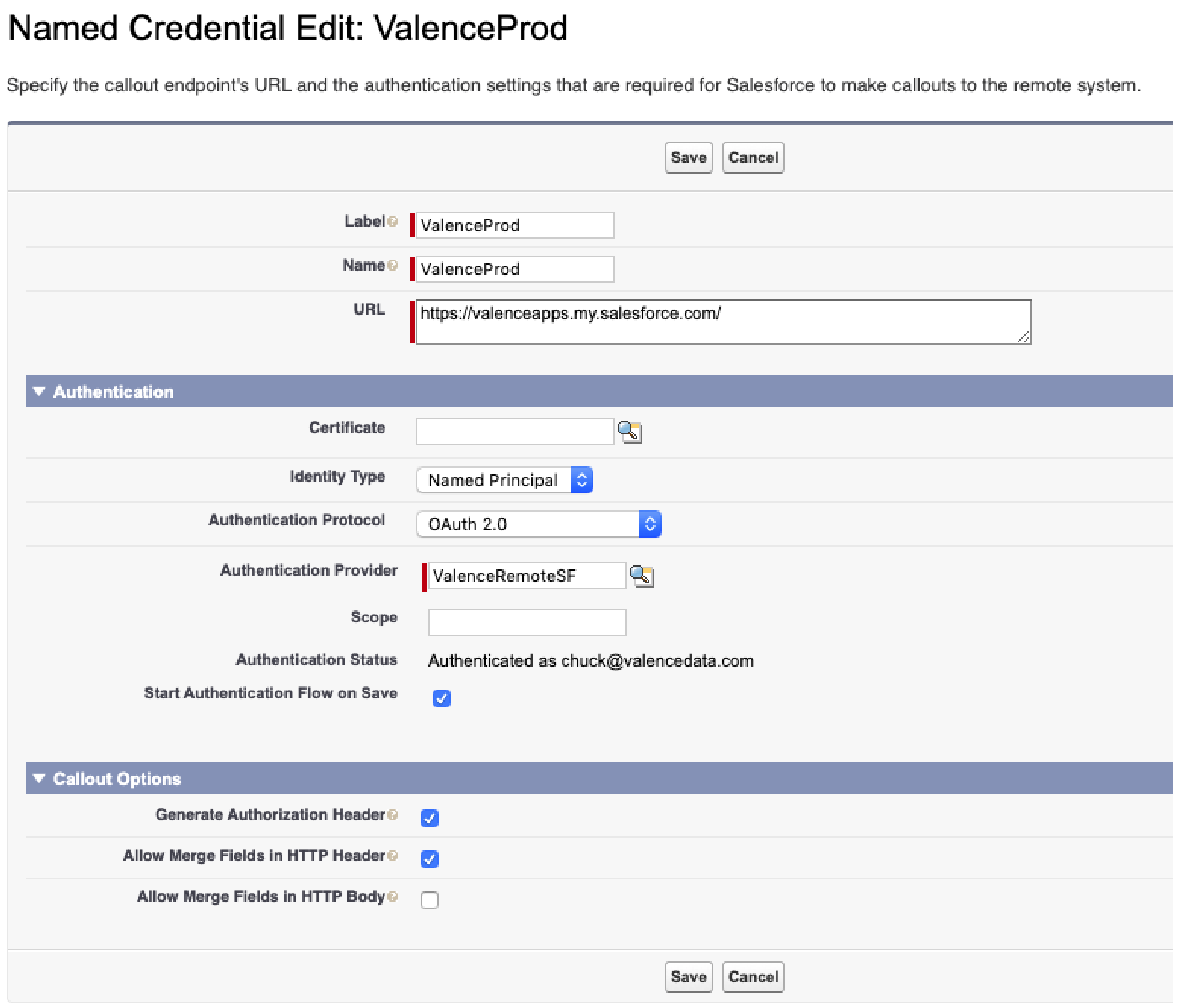
Now *it's time to set up a Named Credential*.

5.16.5 Create a Named Credential For Each Remote Salesforce Org

Named Credential is a standard Salesforce feature that is a container for authentication details. Each record holds a URL and the necessary credentials to go talk to that URL.

You will need one Named Credential record for each remote Salesforce org you will be talking to.

Here is an example Named Credential:



Named Credential Edit: ValenceProd

Specify the callout endpoint's URL and the authentication settings that are required for Salesforce to make callouts to the remote system.

Save Cancel

Label	ValenceProd
Name	ValenceProd
URL	https://valenceapps.my.salesforce.com/

Authentication

Certificate	
Identity Type	Named Principal
Authentication Protocol	OAuth 2.0
Authentication Provider	ValenceRemoteSF
Scope	
Authentication Status	Authenticated as chuck@valencedata.com
Start Authentication Flow on Save	✓

Callout Options

Generate Authorization Header	✓
Allow Merge Fields in HTTP Header	✓
Allow Merge Fields in HTTP Body	☐

Save Cancel

1. For the URL you need to use your My Domain URL for the remote Salesforce org
2. For Identity Type selected “Named Principal” or “Per User” (most people used “Named Principal”; to understand “Per User”, read the Named Credential documentation)
3. For Authentication Protocol choose “OAuth 2.0”
4. For Authentication Provider select an Auth Provider you set up in the previous step
5. Leave Scope blank (will use the default values from your Auth Provider)
6. Check off “Start Authentication Flow on Save”; after saving the record you will be redirected to log into the remote org to grant access for this org to interact with it
7. Make sure “Generate Authorization Header” is checked
8. Make sure “Allow Merge Fields in HTTP Header” is checked
9. It doesn’t matter if “Allow Merge Fields in HTTP Body” is checked or unchecked, we recommend leaving it unchecked to help prevent leaky credentials

5.16.6 What You End Up With When You're Done Building Authentication

When you're all done configuring your authentication, you will have:

- Nothing defined in any of your remote orgs (all of this setup was in the controlling org)
- A Named Credential record in your controlling org for each remote org you want to talk to
- An Auth Provider that sits underneath the Named Credentials and tells them where to go (You can use the same Auth Provider for multiple Named Credential records)
- A Connected App underneath the Auth Provider that facilitates the handshake between orgs

5.16.7 Finally, Some Valence Stuff

Now that you have done the hard part (setting up your authentication), the Valence side of things is pretty trivial.

You can build as many Links as you like, configured to move data between these environments.

In order to accomplish this you will be using a combination of two Adapters that come packaged in Valence: the "Local Salesforce" and "Remote Salesforce" adapters.

Let's say you have **Org A** with Valence installed (aka your "controlling org"), and then two other Salesforce orgs: **Org B** and **Org C**. Here are some possible scenarios you could build:

Link	Source Object	Source Org	Adapter Type	Source Named Credential	Target Object	Target Org	Adapter Type	Target Named Credential
Example 1	Account	Org A	Local	No	Account	Org B	Remote	Yes (Org B)
Example 2	Lead	Org B	Remote	Yes (Org B)	Contact	Org C	Remote	Yes (Org C)

You get the idea. You'll use the Local Salesforce Adapter to get data in and out of the org where Valence is installed, and the Remote Salesforce Adapter for other orgs.

You can move data any direction between any combination of orgs.

When building a new Link, select the "Remote Salesforce" Adapter as your Source Adapter and/or Target Adapter. When you do so, you'll be asked which Named Credential to use. Select one and you'll be dynamically shown objects that are defined in that remote org.

That's it. Enjoy!

5.17 How To Stop Infinite Loops In Bidirectional Syncs

5.17.1 Scenario

You're excited to put a sync in place between the Salesforce Account object and the Customer table in your backend accounting software. You build a Link (Link A) that listens for changes on the Account object and sends them to the accounting system, and you build a Link (Link B) that listens for changes on Customer in the accounting system and sends them to Salesforce. Badda boom, bidirectional sync!

But there's a problem: you create a new Account called Acme in Salesforce at 2:00pm, and Link A picks it up and writes it to the accounting system at 2:30pm. Link B runs at 2:45pm and sees a recently-edited Customer record, so it sends it off to Salesforce. Link A runs again, and sees that Acme was updated at 2:45pm, and since the last time Link A ran was 2:30pm, it's a fresh change! Better send that highly-valuable new info over to the accounting system. And so on, and so on, and so on…we've just created a vampire record that will bounce back and forth between these systems until the end of time.

This is a side effect of doing bidirectional delta syncs (between any integrated systems, this isn't Valence-specific), and it's not uncommon. Unfortunately, the more you poke at it and investigate the more complicated you realize it actually is. Let's dig in.

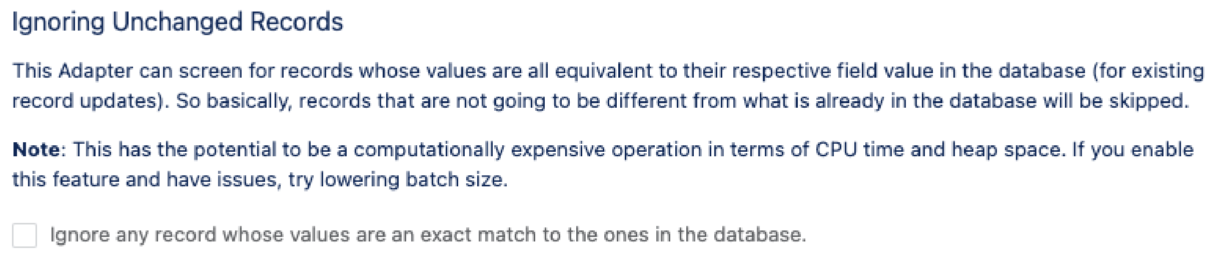
5.17.2 Solutions

Below are some strategies for breaking the loop, and considerations, pros, and cons for each. There are many different ways you can tackle this problem, and that's because there are many subtle nuances to the circumstances in which this problem exists. The right strategy changes depending on if you are doing scheduled Links or realtime Links. It matters if your record transformations are deterministic. It matters what your contention resolution needs are, and how you handle system of record.

The point is, take some time to carefully think about your unique circumstance you are solving for and consider potential edge cases and if your plan covers them. Some of the solutions presented below are just feature toggles, and some are less prescriptive and more about arming you with patterns and ideas.

LocalSalesforceAdapter configuration

When using the LocalSalesforceAdapter as a target, there is a configuration option you can select right after you've chosen which field should be your upsert field:



Ignoring Unchanged Records

This Adapter can screen for records whose values are all equivalent to their respective field value in the database (for existing record updates). So basically, records that are not going to be different from what is already in the database will be skipped.

Note: This has the potential to be a computationally expensive operation in terms of CPU time and heap space. If you enable this feature and have issues, try lowering batch size.

☐ Ignore any record whose values are an exact match to the ones in the database.

This is your "easy button" for breaking the loop. If you turn this on, generally it's all you'll need to do and you can stop worrying about it.

Be aware that using this method is a *little* inefficient, as it will allow a record created in Salesforce to go to the external system, then come back into Salesforce before it is squashed, so that's one extra hop. In the big picture it's usually not a problem, just some extra inbound records that you don't care about and will be marked as ignored.

Warning: This pattern will not work if your Link is not deterministic. 99% of Links are deterministic, meaning the same record with the same incoming field values will produce the same target record. However, as an example, there is one thing people like to have that will cause a Link to be non-deterministic: adding a timestamp for "last-synced". If you are generating a timestamp in a Filter and writing it into Salesforce, then the same record will look different each time it arrives, and this pattern won't work.

Valence Fingerprinting

There is a **Fingerprinting** feature you can enable in the Link Settings for any Link.

When this is active Valence will use hashes of each record and compare traveling records to past records for exact matches. If the exact same record is being delivered twice in a row, the record is blocked. Blocked records disappear from the batch they are in, are not delivered, and no record snapshot is generated. There are two exceptions: if a record is the same as it was before but last time it ran the record failed or was ignored, it is allowed to attempt delivery this time (in case things have changed and it would be a success this time).

Looking back at the scenario we started with, the Account would go through Link A to the external system, back through Link B, then get picked up by Link A but not delivered. So this pattern would involve one more hop than the previous one, but the ultimate result is the same.

This feature is similar to configuring the LocalSalesforceAdapter as described above, but differs in the following ways:

- Records are blocked, not ignored (ignored records show up in Sync Events with their ignore reason)
- Works for any Link going any direction (configuring LocalSalesforceAdapter only works for inbound to Salesforce, but of course breaking one side breaks the loop)

Warning: The same warning about deterministic records applies here as well (see the previous warning for more detail).

Record-based Marker

For remaining patterns including this one, we're moving away from Valence features into customizations and strategies that you can take advantage of if the previous two options are not a good fit.

A record-based marker is some kind of identifier on the record itself than can be used to make a decision about whether it should be written into the target (or, also if it should be picked up and written back out).

The big advantage of a record-based marker is that it is durable. After the Link run ends we still have access to the marker. This means it's available to any Link at any later point in time, meaning **this pattern works for both scheduled and realtime Links**.

The most common pattern here is some kind of "last touch" field with string values like "Quickbooks" and "Salesforce". All Links writing into Salesforce from Quickbooks assign "Quickbooks" to this field, and all Links writing into Quickbooks assign "Salesforce" to this field. Then, you filter both Links to only pick up changed records that are from the local system (or filter one side, still breaks the loop at the cost of an extra hop).

This works well enough, but it can be a little fragile. You have to make sure that all the ways you can change a record normally (say in Salesforce if someone edits the field from its record page) mark the "last touch" field back to "Salesforce" (in this example) so that it can be picked up when you want it to be.

The easiest way to do this is in a before trigger on that SObject, but you have a challenge: how do you differentiate between the trigger firing because Valence wrote to the SObject, and all other trigger fires? Unfortunately there's no way to tell normally, so we recommend the use of some kind of static flag that you set from a Filter and then check in your Apex trigger to let yourself know this write is a result of a Valence Link.

Tip: If you have automations that fire and create/update other records than the arriving ones, you may want to listen to those and allow them to be emitted back out by Valence, so think carefully about how you design your flag.

Context-based Marker

This is similar to the record-based marker described above, with some key differences. With a context-based marker the implementation is a little simpler, but it **only works for realtime Links**.

You have a static flag somewhere like before (can be as simple as a `static Boolean` somewhere), but you don't mark up the records themselves.

Let's say you have Link A that writes into the Contact object, and Link B that listens to the Contact object using a Flow or Apex trigger.

Link A runs, and some records are on their way inbound to the Contact object. Set the flag with a Filter during the run. Ding! Alert! Valence run in progress.

Then, when the Flow or Apex trigger that drives Link B reacts to the records being written to the database, check the flag. If it's set, ignore these records, don't reflect them outbound if they've just arrived. If false, send them outbound, they didn't come from this execution context.

The reason this doesn't work for scheduled Links is that once your current execution context ends you lose the flag, so it is only effective if you're teeing up records to go outbound in the same context they are arriving inbound.

Tip: If you have automations that fire and create/update other records than the arriving ones, you may want to listen to those and allow them to be emitted back out by Valence, so think carefully about how you design your flag.

5.17.3 Other Bidirectional Sync Considerations

Since we're already talking through the ramifications of syncing two tables in both directions, let's take the opportunity to break down a few more gotchas and words to the wise.

Data Enrichment

Generally we think that a record that just arrived in a system should not be sent right back out the other direction. However, there's an exception to this: if a record is transformed upon arrival, you may want to allow it to reflect right back out.

Here are some potential reasons:

- Some default values are set in some of the fields and those values are worth sending back to the original system
- Identifiers of some kind are generated on this new record
- Sub-records or related records are generated

In these scenarios, you actually want the extra hop so that both systems end up with these enriched fields. Using a pattern like LocalSalesforceAdapter configuration serves you well here.

Duplicates

It is common to set up a bidirectional sync and then start getting duplicates. This is because you're typically writing to each system with the unique identifier from that system. A new record arrives in a system and is created there, then the record is sent back the other direction carrying its new identifier, but because it's a new identifier the upsert back the other direction doesn't find it and inserts a record. So you get two records in the original system, one record in the other system, and the very first record is orphaned (and further edits to it will spawn more duplicates).

The fix here is straightforward. Almost all APIs return new identifiers in response to record creation, so the target Adapter delivering the records needs to do a "writeback". A writeback is a small record, usually just a pair of identifiers, sent back to the original system so that the new identifier is correctly paired with the original source record.

Contention

If two systems can both edit the same record, how do you handle the scenario where both systems have each independently changed the record before a sync occurs? Or, even worse, each system fires their change and both systems lose data.

This can be enormously tricky to completely solve, so most implementations settle for a "good enough" plan. The most common is to just let everything write when it arrives, and hope that syncs and edits don't line up in a way that causes data loss. This is usually fine, honestly, if your syncs are pretty frequent and your edits infrequent (at least edits to the same record).

A slightly more refined approach is an "ignore if newer" behavior. You pass the most recent edit timestamp for a record in the source system as part of the payload, and then check the most recent edit in the target system, and whichever is newer is the winner (resulting in write or ignore). It's not too hard to set up a Filter that does this evaluation and uses the ignore() method on RecordInFlight.

However, this approach treats newer as better, but it's just as likely that both edits are worth preserving. A more accurate approach would be to merge the records together and preserve both edits. As you can imagine merging records can be quite a nuanced operation… best of luck! You've got this.

One more contention resolution strategy is to leverage a "system of record" style of thinking, where one system "owns" a record and that system always wins contention.

System of Record

When we introduce the idea of system of record, we are introducing primacy between our two systems. One system is the golden record, the truth, and the other system should match and if it doesn't the golden record is the correct one.

Most businesses identify a system of record for any given object or table. Maybe your Contacts treat Salesforce as the System of Record, but your Accounts treat your backoffice accounting system as the system of record.

Declaring a system of record for an object can greatly simplify your strategy for resolving contention.

Sometimes, system of record gets a little more granular. Here are some patterns:

- Object-level System of Record: each object is "owned" by one of the participating systems, and any contention favors that system.
- Record-level System of Record: the first system that creates a record "owns" it, and each system holds records that are owned by each system. The system that owns a particular record wins during contention resolution.
- Field-level System of Record: some fields on a record are owned by another system. This is quite common when you are using other tools, like tracking data, to enrich your records. Maybe 10 fields on Account belong to your tracking tool which would win contention only on edits to those 10 fields.

System of record can tie back to everything we've discussed in this document.

One of our early customers was bidirectionally syncing a table in Salesforce with their backoffice system, and used field-level ownership to resolve contention. In addition, if a record arriving in Salesforce only changed fields owned by the source system, no outbound sync record would be sent. So not only did they use it for contention but also for blocking infinite loops.

You could also do the reverse, where you only pick up records to send if certain fields have changed, not just any change.

5.18 Extensions

5.18.1 Introduction

Valence is designed to be extensible, and provide places for custom code and logic to be added. These extensions are gracefully integrated into the UI and behavior of Valence so that Users can configure and work with them as easily as they work with standard Valence features.

Deciding to add custom code does not compromise your ability to still configure, control, design, and observe everything using a nice, clean UI.

Valence uses Apex Interfaces to interact with extension Apex classes. These interfaces define a contract between the Valence framework and custom extensions that allows code developed at different times in different orgs to still work together.

In order for your custom code to be usable by Valence, you implement whichever interfaces match what you are trying to do.

As a design methodology, we have used many smaller interfaces to define discrete slices of functionality. You will almost certainly be implementing several interfaces each time you write a custom extension.

By combining your registration cMDT record and the interfaces your code implements the Valence framework can dynamically detect the presence of your extension and expose sophisticated behavior to end users.

Extension Requirements

In order to create your own custom extension, you need to do two things:

1. Create a **global** Apex class that *implements one or more Valence extension interfaces*.
2. *Register your extension* with the installed Valence app by creating a custom metadata type record in the appropriate table. This tells Valence that your extension exists and where to find it.

Types of Extensions

A Valence extension is an Apex class that can be hooked into Valence and called at specific points in time to run custom code. There are two types of Valence extensions: Adapters and Filters.

Adapters

An Adapter is an Apex class that knows how to talk to some external system or data store. Valence can make the connection to another system, but without an Adapter we wouldn't know how to exchange information with that system. It's as if Valence places a telephone call to someone that speaks Latin, and the Adapter translates the Latin into English.

Every *Link* defines two Adapters that it will use: a **source** Adapter to get data from, and a **target** Adapter to send data to. Some Adapters can only be used as a source, some only as a target, and some can be either.

Hint: You can use one Apex class and implement many interfaces (source + target), or split things up and have different Apex classes for source and for target. It's up to you! Just remember that you need *one* cMDT registration record for *each* Apex class you create. The registration record is how Valence locates your Apex class to instantiate it.

Filters

A Filter is an Apex class that processes a data record as it moves through the Valence engine. It is an opportunity to observe and possibly manipulate records as they go by. Any number of Filters can be attached to a *Link*, and the order in which they fire (and any configuration of them) is controlled by the User setting up the Link.

Here are some example use cases for Filters:

- Adding a constant value to records going by.
- Manipulating date/time strings to be friendly to the target system.
- Filtering out records that should be ignored and not processed.
- Validating records for custom business logic that would add a warning or an error to a record.
- Transforming record shape or field values.

Valence includes some Filters out of the box for you to use.

At a minimum you will write an Apex class that implements the *TransformationFilter* interface. Filters can be very, very simple or they can be quite sophisticated.

Filters are configured within a Link as a chain, where each is processed in turn and has access to whatever modifications earlier filters applied to the records (see: Intercepting Filter Pattern). This means Filters can be cumulative and build on the results of previous Filters. Users set the order in which Filters run as part of the configuration for a Link.

You can write sophisticated Filters that offer a *Configuration Component* for admins to configure your Filter behavior. We eat our own dog food, so if you've seen the UI for configuring the RelationshipFilter, for example, then you've seen *ConfigurablePerMappingFilter* with a custom UI component in action.

5.18.2 Interacting with Valence from your Custom Extension

Valence is an orchestration engine with lots of moving parts. When you build a custom extension, you are adding an additional cog to that machine. You tell Valence where your cog fits into the machine by implementing interfaces. These interfaces determine when your code will run, what values are passed to it, and what is expected back from it.

To facilitate sophisticated data exchange, a collection of global Valence classes are part of the method signatures of these interfaces.

Valence Classes

Class	Description
AdapterException	Special Exception class that we encourage you to use in your Adapters to indicate that something has gone wrong that cannot be recovered from.
CSVReader	Utility class that makes it easier to parse raw CSV data
FetchStrategy	FetchStrategy allows your source Adapter to tell Valence how best to ask your Adapter for records. It is an example of the Strategy Pattern https://en.wikipedia.org/wiki/Strategy_pattern.
Field	The Field class represents a property that a record may have. It is analogous to a table column, or in Salesforce an object field.
Field Path	FieldPath represents a the location of a Field within a schema tree, and is used to traverse that tree to find the right value.
FilterException	Special Exception class that we encourage you to use in your Filters to indicate that something has gone wrong that cannot be recovered from.
JSONParse	Utility class that makes it easier to parse raw JSON data
LinkContext	This is a class that is full of information that might be useful to your Adapter or Filter while it is executing. It is an example of the Context Object pattern.
Mapping	Mapping is a special Apex class that gives Adapters and Filters info about the mappings a user has defined for the Link.
Property Node	PropertyNode helps us to interact with the data values on a record, and allows reading and writing using property path notation.
RecordInFlight	RecordInFlight represents a single record as it moves through the Valence framework. RecordInFlight holds not just the record properties but also metadata such as errors and warnings associated with the record.
Table	The Table class represents a possible source or target for a Link. It is analogous to a database table or Salesforce object.

Adapter Interfaces

Interface	Source/Target	Description
ChainFetchAdapter	Source	Implement if your Adapter is fetching data from a system that cannot predict in advance how many records (or how many batches) will be needed to retrieve all the records. This interface allows you to alternate record fetches with record processing indefinitely until all source records are exhausted.
ConfigurableSourceAdapter	Source	Implement if your Adapter is user-configurable when used as the **source** of records on a Link.
ConfigurableTargetAdapter	Target	Implement if your Adapter is user-configurable when used as the **target** of records on a Link.
DelayedPlanningAdapter	Source	Implement if your Adapter needs a bit of real-world time between when it is asked for data and when it is ready to serve that data.
LazyLoadSchemaAdapter	Either	Implement if your Adapter can load part of its schema independently and later in time. Helps to improve performance working with large schemas.
NamedCredentialAdapter	Either	Implement if your Adapter supports using a NamedCredential as its source of endpoint and credential info. Almost every Adapter will likely implement this interface.
SchemaAdapter	Either	Implement if your Adapter can describe its schema. This interface is not required, and in its absence Valence will still do what it normally does: discover record shape dynamically as records are processed.
SourceAdapterForPull	Source	Implement if your Adapter can be the source of records when fetching records from an external system. This is the most common Adapter variant.
SourceAdapterForRawDataPush	Source	Implement if your Adapter is parsing raw data (like a JSON packet) as the beginning of a Link run. You'd use this if you had real-time data packets arriving, for example via API into the Salesforce org.
SourceAdapterForSObjectPush	Source	Implement if your Adapter wants to convert sObject records into something that can be sent to another Adapter. Very unlikely that you would use this one.
SourceAdapterScopeSerializer	Source	Implementing this interface opens up several features for Users, such as parallel batch processing and replaying failed batches on Links where your Adapter was the source.
TargetAdapter	Target	Implement if your Adapter can be the endpoint of a Link, i.e. the place where records are sent at the end of processing.

Filter Interfaces

Interface	Description
ConfigurablePerLinkFilter	This interface allows a Filter to have multiple configurations applied to it on a given Link that are independent of any particular Mapping.
ConfigurablePerMappingFilter	This interface allows a Filter to have a configuration applied to it that is different for each Mapping on the Link. This allows an admin to make the Filter behavior differently for each Mapping, or skip certain Mappings.
LinkSplitFilter	Implement this interface if your Apex class is capable of helping to direct specific RecordInFlight instances to their matching Link Split destinations.
SchemaAwareTransformationFilter	This interface allows your Filter to describe its impact on the source and target schemas of records, and will better help Users understand what your Filter does.
TransformationFilter	This is the primary interface to implement if you are building a Filter. It will allow your Apex class to inspect records that are being processed, and modify them if need be.

5.18.3 Register Your Extension With the Valence app

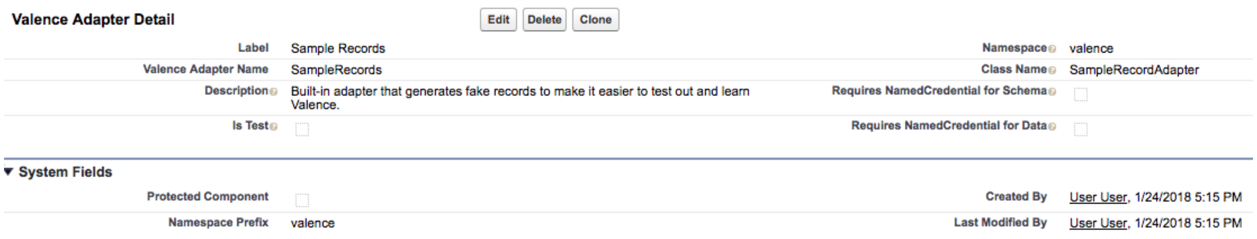
Tip: Registering a custom extensions is only necessary if you are writing the Apex code yourself! If you are simply installing a custom extension that someone else wrote, whomever developed it would have already followed these steps and included the cMDT record in their package alongside their Apex class.

Registering your extension tells Valence that it exists and what it is called, and also how to instantiate your Apex class. It is also an opportunity to specify some additional information about how your extension should be (and should not be) used.

To create this record go to your **Setup** menu and type “custom metadata types” in the Quick Find. Click “**Manage Records**” next to the custom metadata type you’re going to add a record to.

Register a Custom Adapter

Telling Valence about your custom Adapter is done by making a new record in the **valence__ValenceAdapter__mdt** custom metadata type.



Valence Adapter Detail

Label	Sample Records	Namespace	valence
Valence Adapter Name	SampleRecords	Class Name	SampleRecordAdapter
Description	Built-in adapter that generates fake records to make it easier to test out and learn Valence.	Requires NamedCredential for Schema	☐
Is Test	☐	Requires NamedCredential for Data	☐

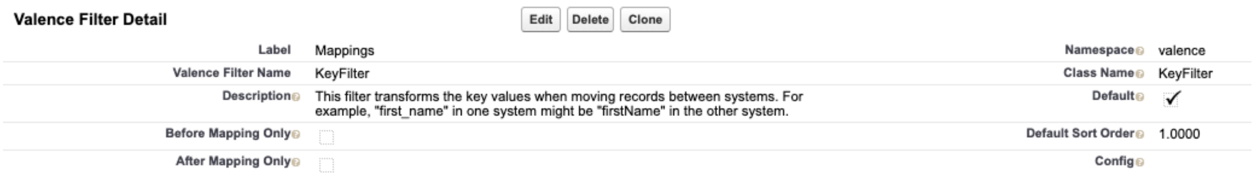
▾ **System Fields**

Protected Component	☐	Created By	User User, 1/24/2018 5:15 PM
Namespace Prefix	valence	Last Modified By	User User, 1/24/2018 5:15 PM

API Name	Label	Description
DeveloperName	Name	The API name of this record, used whenever this record is edited or retrieved programmatically.
MasterLabel	Label	The user-friendly display label for this record.
ClassName__c	Class Name	The Apex class name of the Adapter class that will be instantiated.
Namespace__c	Namespace	The namespace of the Adapter class that will be instantiated. **Does not** have to be the same namespace as this cMDT record. If you are not packaging the Adapter, you can leave namespace blank.
Description__c	Description	A description of the purpose of this Adapter. Shown to users in the Valence UI.
Requires-Named-Credential-ForSchema__c	Requires NamedCreden-tial for Schema	Checked if this Adapter must be given access to a NamedCredential in order to return its Schema.
Requires-NamedCreden-tialForData__c	Requires NamedCreden-tial For Data	Checked if this Adapter must be given access to a NamedCredential in order to exchange records with an external system.
IsTest__c	Is Test	Check this if this Adapter is used only for Apex tests, in which case you'll also likely want to check the **Protected Component** field as well.

Register a Custom Filter

Telling Valence about your custom Filter is done by making a new record in the **valence__ValenceFilter__mdt** custom metadata type.



Label	Mappings	Namespace	valence
Valence Filter Name	KeyFilter	Class Name	KeyFilter
Description	This filter transforms the key values when moving records between systems. For example, "first_name" in one system might be "firstName" in the other system.	Default	✓
Before Mapping Only	☐	Default Sort Order	1.0000
After Mapping Only	☐	Config	

API Name	Label	Description
Developer-Name	Name	The API name of this record, used whenever this record is edited or retrieved programmatically.
MasterLabel	Label	The user-friendly display label for this record.
ClassName__c	Class Name	The Apex class name of the Filter class that will be instantiated.
Namespace__c	Namespace	The namespace of the Filter class that will be instantiated. **Does not** have to be the same namespace as this cMDT record. If you are not packaging the Filter, you can leave namespace blank.
Description__c	Description	A description of the purpose of this Filter. Shown to Users in the Valence UI.
Default__c	Default	Checked if this Filter should automatically be included on any new Links that are created by Valence users.
DefaultSortOrder__c	Default Sort Order	If this Filter is used as a default, the starting Sort Order value that this Filter will have as part of a Link. This default can be overridden per Link.
BeforeMappingOnly__c	Before Mapping Only	Check this box if your Filter works with the original properties of a record and only makes sense to run before Mappings have moved values over to the properties side of RecordInFlight.
AfterMappingOnly__c	After Mapping Only	Check this box if your Filter works with the properties of a record and only makes sense to run after Mappings have moved values over to the properties side of RecordInFlight.

5.18.4 Packaging

Valence extensions, as well as configuration/customization done in the Valence UI, are packageable and deployable. Because extensions are Apex classes and registration is done with custom metadata types, you can create a Salesforce package for distribution that includes both registration records and the Apex classes they refer to.

At a minimum your package should include an Apex class and the matching cMDT record that registers your Apex class.

You can even go beyond this and package not just custom code but also Links, Mappings, etc. All Valence configuration is done with custom metadata types, which are themselves packageable.

Example:

Note: Let's say you were a domain expert in the Real Estate business and you had a managed app that installed a custom object Property__c with a bunch of custom fields on it.

With Valence you could create a package that included:

- A custom Adapter that knew how to talk to MLS servers (external servers that store listing information about real estate properties).
- A pre-built and configured Link that pulled listing information out of the MLS server and put it into your custom Property__c object.
- Pre-built mappings for all the usual fields that people need from the MLS to Property__c.

Not only could you package all this and install it for customers alongside your package, those customers could then further customize the Links and Mappings if they wanted to.

5.19 Property Path

One of the most important parts of working with *RecordInFlight* instances is making it as easy as possible to read from and write to the record, without losing the ability to do complex and sophisticated behaviors.

To solve this we have created a specific syntax that we call a "property path", which is inspired by XPath and JSONPath. A property path describes one or more locations within a record's data properties.

Property path notation is intuitive:

```
recordInFlight.getPropertyValue('Parent.Name'); // returns a single Name
recordInFlight.getPropertyValues('Contacts[*]'); // returns a collection of rich objects
recordInFlight.getPropertyValues('Contacts[*].FirstName'); // returns a collection of␣
↪FirstName values
recordInFlight.getPropertyValues('Contacts[0,1].FirstName'); // returns a collection␣
↪containing the FirstName value of the first and second Contacts
```

There are a number of methods on *RecordInFlight* that allow you to use a property path to read or write information. For advanced use cases and maximum flexibility, you can skip those methods and work with the backing data store directly: *Property Node*

5.19.1 Two Formats of Property Path

The official, formal definition of a property path is a List<String> where each element is one of two types: a "field" or a "selector". A "selector" follows a list and specifies *which* items from that list we want to work with.

```
// field, selector, field
['"Contacts"','*','"FirstName"']
```

We refer to this format as a "normalized" property path, and it is what the engine works with internally.

Since the normalized property path format is a unwieldy to work with, there is a "concise" format of property paths:

```
Contacts[*].FirstName
```

Selectors are bracketed and follow their list, and dots are used as delimiters. Anywhere property path is used in Valence, you can choose whether to use the concise or normalized format. If you choose concise, we'll normalize it under the hood.

The syntax for a property path is different when reading data and when writing data. That's because when writing data we also want to include additional instructions for how to handle intermediary layers and how exactly we want to manipulate the destination nodes.

5.19.2 Using Property Path To Read Data

When reading data, field elements are the same as writing data but list selectors differ.

Operator	Label	Description
*	Get All	Reads from every possible node in this list
<number> (, <number>)	Index	Read from each specified index (indexes start at zero)

As you can see, reading is pretty simple. Either you grab all the list items or you can specify certain ones you're interested in.

5.19.3 Using Property Path To Write Data

When writing data, field elements are the same as reading data but list selectors differ.

Operator	Label	Description	Restrictions
*	Copy For Each	Each *existing* item in the target list will receive its own copy of the data value from the source side.	Cannot be used as final path element
...	Distribute	When you have a source list and want to give each *existing* target item one thing from the source list (the opposite of 'Copy For Each').	Cannot be used as final path element
:	Line Up	Preserve the source item list position when writing over to the target; this is the one you'll use the most often.	
+	Add To	Add values to what is already in the target list (initializing if empty).	
<number> (, <number>)	Index	Write to each specified index (indexes start at zero).	

5.20 Configurability

5.20.1 Overview

Valence is a sophisticated engine and allowing users to tailor it exactly how they need it is intrinsic to its value.

Each extension should be designed with these users in mind, and built from the ground up to have flexibility where it needs to.

We've provided quite a bit of framework to support your development effort, and you can focus on what's unique and special about your extension while mostly staying out of boilerplate and orchestration.

Both Adapters and Filters can be user-configurable, and as you'd expect you declare support for this functionality by implementing certain interfaces:

Adapter interfaces:

- *ConfigurableSourceAdapter* - if you're a source Adapter
- *ConfigurableTargetAdapter* - if you're a target Adapter

Tip: You might implement both of these if your Adapter can both send and receive data.

Filter interfaces:

- *ConfigurablePerLinkFilter* - configurations that are agnostic of any particular mapping
- *ConfigurablePerMappingFilter* - configurations that depend on a mapping to provide a value or field

5.20.2 Apex-side

Once you implement one of the configuration-related interfaces, you'll see some standardized methods be required. They fall into three groups:

- Asking your extension for information about how it can be configured
- Asking your extension to explain what a particular configuration does
- Handing configuration details to your extension at runtime

You will never be responsible for storing configuration data between runs, or knowing where to go find it. At runtime Valence hands you exactly the configuration information you need, when you need it.

Tip: We always want users to be in the driver's seat and understand what their engine is doing. Explaining your configurations is a really important part of helping users to understand the effects of their decisions. Take time to design a thoughtful and clear response to your implemented explain method.

Listing 16: Example Explain Method

```
public String explainFilterConfiguration(String configurationData) {

        try {
                Configuration config = buildConfiguration(configurationData);

                String message = 'A source field called <strong>{0}</strong> will be␣
↪added to each record that is actually a sub-object built as follows:<ul>{1}</ul>';

                List<String> entries = new List<String>();

                for(ConfigField configField : config.fields) {
                        entries.add(String.format('<li><strong>{0}</strong> (Value comes␣
↪from <strong>{1}</strong>)</li>', new List<Object>{
                                configField.fieldName,
                                String.join(configField.sourcePath, '.')
                        }));
                }

                return String.format(message, new List<Object>{
                        config.resultName,
                        String.join(entries, '')
                });
        }
        catch(Exception e) {
                return '<p class="slds-theme_error">The configuration for this Filter is␣
↪malformed.</p>';
        }
}
```

5.20.3 Client-side

We've briefly discussed how to consume configuration data at runtime in your Apex class, but how does the User see, create, and modify configuration data for your extension?

Valence offers you the choice of either of these approaches:

1. *Handing Valence a structured representation* of your configuration that Valence will build into a simple web form and allow Users to populate.
2. *Defining a custom Lightning component* that Valence will instantiate and embed in its user interface for Users to interact with when they choose to configure your extension.

Both are equally viable. If your configuration is nuanced or complex, or needs to pull extra data in (such as populating picklist fields) you'll want to go with a custom component.

The user will be able to configure your extension differently for each Link (or even each Mapping in the case of certain Filter extensions). Valence will store the configuration data provided by the User, and hand it to your Apex class at the appropriate moments.

5.20.4 Configuration Hierarchy

There are special fields on Adapter__mdt and Filter__mdt that allow you to specify additional configuration values. The format should be a JSON object with key value pairs (nested objects are fine).

You can use this layer of configuration to set up default values or configuration values that are independent of any particular Link.

During execution, Valence will merge any configurations that were defined by the user at the Link level with the configurations on these two custom metadata types, with the Link-level configurations overriding the independent values in case of a conflict (this allows you to have admin users override your default values per-Link if need be).

The configuration data handed to your extension at runtime will be the merge result of these two different layers of configuration data.

5.21 Configuration Structure

Configuration structure allows you to define an expected shape of your configuration and Valence will handle creating the UI and getting values from users. It is mutually exclusive with using a custom Lightning component.

Each Valence interface relating to configuration has a method that asks for your "configuration structure". You return null if you're not using configuration structure, or a special JSON string if you are.

Valence will use the JSON you return from this method to dynamically instantiate a form for the user to fill out.

At the bottom of this page you'll read about a helper class that can build this JSON for you, but let's begin by exploring the JSON shape so you understand how it works.

5.21.1 Expected Structure

The shape Valence expects has two properties:

```
{
    "description" : "A description for the user to read about how to go about␣
↪configuring your extension. Shown at the top of the screen.",
    "fields" : [
        { ... a field object ... },
        { ... another field object ... }
    ]
}
```

description [required]

Instructions that will be shown to the User alongside a form. These instructions should explain overall what the User needs to do to fill out the configuration fields appropriately, and anything else they might need to know or be aware of.

fields [required]

An array of fields, each one representing a value that a User could potentially fill in.

5.21.2 Field Shape

A field object represents a single value that the user might set.

```
{
    "name" : "short name for your field",
    "attributes" : {
        "maxlength" : 15
    },
    "componentType" : "lightning:input"
}
```

name [required]

The field name for this value. This is important, and required. This is the name we will serialize as the key against whatever value the user provides.

attributes (optional)

These are injected straight into the component when it is instantiated. Any attributes that are defined on the componentType can be set here (with the exception of **value**). For example, you could set a **maxlength** or perhaps some validation. Using this in combination with componentType lets you use any base or custom component you might need, and send appropriate settings to that component.

componentType (optional)

By default we will render a lightning:input base component for the user to interact with. You can override this value and render some other base component, or even custom components if you really wanted to.

5.21.3 Configuration Structure Example

Here's an example of a schema configuration you could return to Valence.

```
{
    "description" : "This is a configuration that offers you some choices about how this␣
↪adapter is going to prepare your breakfast.",
    "fields" : [
        {
            "name" : "eggPreference",
            "attributes" : {
                "label" : "How do you like your eggs?"
            }
        },
        {
            "name" : "baconPreference",
            "attributes" : {
                "label" : "Do you want bacon on the side?",
                "type" : "checkbox",
                "checked" : true
            }
        }
    ]
}
```

5.21.4 Configuration Structure Builder

Valence comes with a fluent builder class to help you construct a configuration structure to return to Valence.

Definition

```
/**
 * Makes it easier to set up dynamic configurations for Valence extensions.
 */
global with sharing class DynamicUIConfigurationBuilder {

        global static DynamicUIConfigurationBuilder create(String description);

        global static DynamicField createField(String name);

        global DynamicUIConfigurationBuilder addField(DynamicField newField);

        global String finalize();

        global class DynamicField {
```

(continues on next page)

(continued from previous page)

```
                global DynamicField addAttribute(String key, Object value);

                global DynamicField withComponentType(String componentType);
        }
}
```

Example Usage

```
public String getSourceConfigurationStructure(valence.LinkContext context) {

        return valence.DynamicUIConfigurationBuilder.create('This is a configuration␣
↪that offers you some choices about how this adapter is going to prepare your breakfast.
↪')
                .addField(
                        valence.DynamicUIConfigurationBuilder.createField('eggPreference
↪')
                        .addAttribute('label', 'How do you like your eggs?')
                )
                .addField(
                        valence.DynamicUIConfigurationBuilder.createField(
↪'baconPreference')
                        .addAttribute('label', 'Do you want bacon on the side?')
                        .addAttribute('type', 'checkbox')
                        .addAttribute('checked', true)
                )
                .finalize();
}
```

5.22 Configuration Component

If you have a more complex configuration for your extension, we've got you covered. You can write a custom Lightning component and the Valence UI with instantiate it, pass configuration data back and forth with it, and save that configuration data back to the server for you. Using a Lightning component is mutually exclusive with using a configuration structure.

Each Valence interface relating to configuration has a method that asks for your "configuration Lightning component". You return null if you're not using a custom component, or the fully-qualified name of your component if you are.

Both Aura and Lightning Web Components are supported (but do yourself a favor and go with LWC). In either case, you always need to return the name in the Aura-style naming convention.

Valid Names

`c:MyAuraComponentConfigurator` `c:myLightningWebComponentConfigurator` `mynamespace:myLightningWebComponent`

Invalid Names

`c-my-lightning-web-component-configurator` `c:my-lightning-web-component-configurator`

Note: Aura components tend to start with an uppercase letter; LWCs always start with a lowercase letter.

Tip: If you are getting a red error message about component creation in the UI when you are expecting to see your configuration component, you probably got the name wrong in your return string!

5.22.1 Interacting With Your Valence Container

Your component is instantiated inside of a smart container that will provide you with some useful context information, manage your configuration state (is dirty / discard changes), and allow users to persist your configuration data back to the server (save changes).

Context

Much like your Apex class receives a LinkContext, your component will receive information about what is going on around it. Define properties with these names and they will be set by the container.

Link

Useful information about the Link that is being configured. Includes things like the source and target tables, the name of the Link, the adapters involved, etc.

Configuration

This is your configuration that will be stored in the database and given to your extension during Link runtime. Valence serializes this object as a JSON string and that string is what you are given in the setConfiguration() Apex method during Link execution.

For the purposes of your custom UI, you do not have to worry about getting an existing configuration out of the database, or persisting a new or updated one back to the database. This is all handled by Valence. If an existing configuration exists, it will be passed to your component at instantiation inside this attribute, otherwise you'll be given an empty object. Modify this object to your heart's content, and when the user eventually clicks the Save button your configuration will be serialized and stored.

Schema

Schema information about this Link (full list of possible source fields, full list of possible target fields). Because fetching schema can be slow, **schema is set asynchronously**. It will be set with a blank value when your component is constructed, and then updated several seconds later with actual schema data. Be sure to check for that blank value and don't do anything with it until real data is populated.

Mapping

If this is an Filter extension that implements *ConfigurablePerMappingFilter*, this property will also be set with information about the specific mapping your configuration will be attaching to.

Events

There is one event the container listens for from your component.

updateconfig

Fire this event whenever your configuration is modified, even if it's still a work in progress. Use isValid to share your opinion about whether the configuration as it stands is acceptable.

Event name: `updateconfig` Param: `newValue` - a javascript object that is the entire configuration Param: `isValid` (optional) - true/false for whether this configuration should be allowed to be saved to the server (enables/disables the Save Changes button)

5.22.2 Subclassing the Valence Configurator Component

We hate boilerplate code if you haven't noticed. Even though interacting with your Valence container is straightforward, we maintain a Lightning Web Component service component that you can extend to make it even easier. Unfortunately Salesforce does not currently allow you to extend a Lightning Web Component from a namespace other than `c` and `lightning`, so rather than include it in the package we have it on GitHub and you'll need to pull it down and put it into your org.

```
import ValenceUIConfigurator from 'c/valenceUIConfigurator';

export default class MyAwesomeConfigurator extends ValenceUIConfigurator {
```

If you extend our component, it will handle working with the context properties, expose some lifecycle functions for you to take advantage of, and generally make you go "ahhh, yes".

Abstract Methods

There are two methods you must implement in your subclass.

getDefaultShape()

Return a starter object that has some reasonable defaults for your configuration.

```
getDefaultShape() {
        return {'eggPreference' : null, 'baconPreference' : false};
}
```

computeValid()

Calculate if the current configuration values look good to you and should be allowed to be saved to the server.

```
computeValid() {
        return !!this.configuration.eggPreference; // check if eggPreference is a truthy␣
↪value and return a boolean true/false
}
```

Lifecycle Methods

There are a number of methods you can add to your subclass to be notified when each of the context properties are set. This is especially useful for `schema`, since it is loaded asynchronously and set more than once.

They have no parameters, simply check the property itself to use the new value.

```
onSetLink()
onSetSchema()
onSetMapping()
onSetConfiguration()
tweakConfiguration() - special lifecycle method allowing you to make changes to the␣
↪configuration right before it is kicked up to the container
```

Example Usage of onSetSchema()

```
fieldChoices = []; // options for the lightning-combobox where the user picks the field␣
↪they want to use

/**
 * Set up our fieldChoices whenever we are given schema
 */
onSetSchema() {

        if(!this.schema) {
                return;
        }

        // set up selection options for the field
        this.fieldChoices = [];
        Object.values(this.schema.Source.children).forEach((node) => {
                this.fieldChoices.push({'value' : node.field.fieldName, 'label' : node.
↪field.fieldLabel});
                // note: we deliberately ignored any nested schema fields as they are␣
↪unlikely to usable for what we're doing here
        });
        this.fieldChoices.sort((a, b) => a.value.localeCompare(b.value));
}
```

Example Usage of tweakConfiguration()

It's not unusual for the format of the configuration that works for the Apex class to not work that well in the user interface when working with Lightning components. One way to handle this is to transform the `configuration` property to suit your needs in the interface. This hook lets you transform it back so you don't persist a format or extra keys you didn't want to. This example comes from our open-source Object Builder Filter.

```
/**
 * Because combobox has to work with a string value and sourcePaths are arrays, we␣
↪enrich each configuration record with a flattened path
 */
onSetConfiguration() {
        this.configuration.fields = this.configuration.fields.map(field => Object.assign(
↪{'flattened' : field.sourcePath.join('::')}, field));
}

/**
 * This is called just before sending the configuration up the chain. We strip out the␣
↪extra key we added to each `fields` entry.
 */
tweakConfiguration() {
        return {
                'resultName' : this.configuration.resultName,
                'fields' : this.configuration.fields.map(field => {
                        return {'fieldName' : field.fieldName, 'sourcePath' : field.
↪sourcePath};
                })
        };
}
```

Utility Methods

Here are the utility methods that exist in the superclass you can invoke to help out.

configUpdated()

Call this method whenever you have made changes to the configuration property. It lets the container know and also invokes the computeValid() method.

trackChange()

You bind this directly to the `change` event on your form components so that you don't even need to do anything with them in your controller, but their value will still be set in the configuration.

Be sure to set the `name` attribute on the form component to the exact configuration property you want it to write to.

```
<lightning-input type="checkbox"
                                 checked={configuration.baconPreference}
                                 name="baconPreference"
                                 label="Do you want bacon on the side?"
                                 onchange={trackChange}></lightning-input>
```

debounceChange()

Same as trackChange(), but it debounces the input first to smooth it out. Preferred for fields where the user is typing the input.

You bind this directly to the `change` event on your form components so that you don't even need to do anything with them in your controller, but their value will still be set in the configuration.

Be sure to set the `name` attribute on the form component to the exact configuration property you want it to write to.

```
<lightning-input value={configuration.eggPreference}
                                 name="eggPreference"
                                 label="How do you like your eggs?"
                                 onchange={debounceChange}></lightning-input>
```

5.23 Writing Test Coverage

Everyone's favorite subject: test coverage! Writing test coverage for your Apex classes can be a bit of a pain, but hopefully this page will get you set up for success.

We'll break down how to get your hands on different things you need, and provide sample code for both Adapter and Filter test coverage.

5.23.1 Common Valence Classes

LinkContext

You will need an instance of *LinkContext* in just about every test. Luckily, they're super simple to construct.

```
valence.LinkContext context = new valence.LinkContext();
context.linkTargetName = 'Account';
context.testingMode = true;
// etc etc
```

You can usually just set whichever properties you know you are going to depend on in the code you are exercising.

Mapping

Often you're constructing *Mapping* instances because you want to set the `mappings` property on LinkContext.

Listing 17: Apex class: ValenceTestUtil

```
global static Mapping createTestMapping(String name, String conciseSourcePropertyPath,␣
↪String conciseTargetPropertyPath, String configuration);

global static Mapping createTestMapping(String name, List<String>␣
↪normalizedSourcePropertyPath, List<String> normalizedTargetPropertyPath, String␣
↪configuration);
```

The keys in the `mappings` property should match the names of each mapping.

```
valence.LinkContext context = new valence.LinkContext();
context.mappings = new Map<String, valence.Mapping>{
        'one' => valence.ValenceTestUtil.createTestMapping('one', 'first_name',
↪'FirstName', null),
        'two' => valence.ValenceTestUtil.createTestMapping('two', 'last_name', 'LastName
↪', null)
};
```

RecordInFlight

RecordInFlight is the last of the four most-common Valence classes you'll need to work with in your tests.

Listing 18: Apex class: ValenceTestUtil

```
global static RecordInFlight createTestRecordInFlight(Map<String, Object> original, Map
↪<String, Object> properties);
```

Use the static method above if you want to build RecordInFlight instances that look like they've already been processed (i.e. have properties set already).

If you just want a basic RecordInFlight instance, you can instantiate it normally:

```
valence.RecordInFlight record = new valence.RecordInFlight(new Map<String, Object> {
↪'first_name' => 'Fred', 'last_name' => 'Johnson'});
```

5.23.2 Sample Code

To see real Adapters and Filters and test coverage for them, check out our open-source extensions:

- https://github.com/valence-filters
- https://github.com/valence-adapters

5.23.3 Mocking API Responses for Adapter Testing

In order to properly test your API callouts you'll want to familiarize yourself with Apex callout mocking. We have some patterns we follow with these that we're happy to share.

Where to Mock

It saves a bit of mental effort if your mocks and tests are in the same class, like this.

```
@IsTest
private class MyAdapterTests implements HttpCalloutMock {

        public HTTPResponse respond(HTTPRequest request) {
                // various responses
        }

        @IsTest static void testBehavior() {
                Test.setMock(HttpCalloutMock.class, new MyAdapterTests());
```

(continues on next page)

(continued from previous page)

```
                // test stuff
        }
}
```

There are a couple pretty good patterns for selecting which mock response is appropriate.

```
@IsTest
private class MyAdapterTests implements HttpCalloutMock {

        private String mockBody;
        private Integer statusCode;

        private static final String FAILURE_RESPONSE = '<response><error><message>
↪Explosions and fire</message></error></response>';

        private MyAdapterTests(String mockBody, Integer statusCode) {
                this.mockBody = mockBody;
                this.statusCode = statusCode;
        }

        public HTTPResponse respond(HTTPRequest request) {
                HttpResponse response = new HttpResponse();
                response.setStatusCode(statusCode);
                response.setBody(mockBody);
                return response;
        }

        @IsTest static void testErrorResponse() {
                Test.setMock(HttpCalloutMock.class, new MyAdapterTests(FAILURE_RESPONSE,␣
↪400));

                MyAdapter adapter = new MyAdapter();

                // test stuff
        }
}
```

```
@IsTest
private class MyAdapterTests implements HttpCalloutMock {

        private static final String FAILURE_RESPONSE = '<response><error><message>
↪Explosions and fire</message></error></response>';

        public HTTPResponse respond(HTTPRequest request) {
                HttpResponse response = new HttpResponse();
                if(req.getEndpoint().startsWith('callout:errorResponse')) { // asking␣
↪for error response
                        response.setHeader('Content-Type', 'application/json;charset=UTF-
↪8');
                        response.setStatusCode(400);
                        response.setStatus('BAD REQUEST');
```

(continues on next page)

(continued from previous page)

```
                        response.setBody(FAILURE_RESPONSE);
                }
                return response;
        }

        @IsTest static void testErrorResponse() {
                Test.setMock(HttpCalloutMock.class, new MyAdapterTests());

                MyAdapter adapter = new MyAdapter();
                adapter.setNamedCredential('errorResponse'); // use named credential␣
↪names to drive which HTTPResponse we get back

                // test stuff
        }
}
```

Gathering Mock Responses

Most of the work setting up your mocking is getting realistic response body values for the API you're writing against. We use either of these approaches:

- Interact with the API using Postman calling various endpoints, and strip whitespace from the response with an online tool so we can make it one line in our test class.
- Add debug logging to your class and actually use it a bit in Valence (or invoke it using temporary static methods), then grab the logs, strip whitespace, and save to your test class.

Be patient gathering these, and comprehensive. Try to get all the variations of error messages, and all the variations of responses you might expect. For example, when mocking a record fetch:

- Have a mock response with no records
- Have a mock response with some records, but enough to fit under the context.batchSizeLimit in your test (set it to something low like 5 for the test)
- Have a mock response with more records than will fit in one batch

It's not unusual for us to have 15-30 different API responses mocked in our test class.

5.24 Creating An Adapter

We're going to walk through how to tackle building a Valence Adapter. What to think about, where to start, what's important.

If you haven't already read the primer on how Valence works overall, definitely *read that first*. Also make sure you have read *Extensions* and *Configurability*.

Ok, up to speed? Let's get to work.

5.24.1 The Beginning

An Adapter's job is to be the liaison between Valence and *External Systems* that can share or accept data. Valence and the Adapter work together as partners to strategize about the best way to tackle a goal, and then collaborate to execute on it. If you want to take a peek at some existing Adapters as you read this article, you can find them on GitHub.

Any Valence extension Apex class is going to implement various Valence-provided interfaces in order to be a cog in the Valence machine. These interfaces are like a contract: they define what Valence will provide to you, and also what Valence expects from you. Each interface you implement is a signal to Valence that you support another facet of the Adapter-Valence partnership; just implementing the interface itself is a way of registering your Adapter for that behavior.

It's always good to start small and then build up from there. The first prototype of your Adapter probably implements authentication, schema, and one source or target interface.

Your Adapter never has to deal with orchestration, or timing, or job state. You don't have to think about what execution context you're in, how to recover the job if it fails, if there's already been DML before your callout or not, or if this is a Queueable or a Batch or synchronous. Valence takes care of all of that. Each time Valence interacts with your Adapter you are given exactly what you need to know, and asked for simple, finite things (like the next batch of records).

An Adapter is a distillation of just the unique business logic that is specific to interacting with each external system. You are sitting on top of 95% of the work of creating an integration already being done, and you are writing the last mile.

5.24.2 Authentication

Every Adapter has to figure out how it is going to handle authenticating with the systems it talks to, so *we've got an extensive writeup* on what that looks like.

Adapter Interface: *NamedCredentialAdapter*

5.24.3 Schema

Almost always, Adapters expose a *Schema* so that admin users can select which table they want to interact with, and also can see what fields exist on each table.

Adapter Interface: *SchemaAdapter*

Valence asks you for schema details but how you go about answering that question is encapsulated in your Adapter and up to you. Here are some viable approaches, increasing in sophistication:

- The external system your Adapter interacts with has a very simple schema, so you've just hardcoded your `getTables()` and `getFields()` methods with the schema and called it a day.
- The external system has a relatively static schema but it's rather large, so you ship a flat file (CSV, JSON, etc) with your Adapter as a static resource and then inspect it at runtime when you are asked for schema info.
- The external system has some kind of reflective or discovery endpoint where you can ask it about its tables and fields, so when Valence asks you for schema details you make an HTTP callout to fetch them.

We always prefer that your schema details be dynamic (option #3) so that when an external system's structure is altered, no coding or file changes are needed before admin users can see and work with those changes. Depending on the API you are working with this may not be possible.

Note: Technically it is possible for an Adapter to have no schema. An Adapter that does not implement SchemaAdapter is assumed to basically have one table that is always the one that is read from or written to. The Adapter can still be used as a source or a target (instead of the admin user picking a table they just pick the adapter and move to the next step). Fields are still needed, of course, to do mappings, but instead of the Adapter reporting the fields Valence will detect them during record flow, or they can be specified using ValenceField__mdt instances.

5.24.4 Direction

The next things to figure out is in what direction can your Adapter move data. Is it an Adapter that reads data from an external system (a "source Adapter")? Does it write to an external system (a "target Adapter")? Does it do both? Whenever an admin user sets up a Link they select an Adapter as their source Adapter from the list of source Adapters registered in the org, and a target Adapter from the list of target Adapters registered in the org.

It's perfectly fine to have one Apex class paired with one cMDT registration record that handles multiple source Adapter behaviors and is also a target Adapter. It's also perfectly fine to split these up into multiple classes if that's what makes sense for your circumstances.

Being a Data Source

If you are a source Adapter, your basic responsibility is to produce *RecordInFlight* instances that represent records from an external system. That's pretty much it! Everything else is just about handling the nuance of accomplishing that.

A RecordInFlight is really just a fancy Map of key-value pairs. You build them from a `Map<String, Object>`, like this:

```
valence.RecordInFlight sample = new valence.RecordInFlight(new Map<String, Object> {
↪'first_name' => 'Tom', 'last_name' => 'Smith', 'opt_in' => true});
```

Most of the work of being a data source is understanding which records are needed, and obtaining them in some way.

Operation

Each RecordInFlight has an `operation`, which is a String value that indicates what should be done with this record. We recommend supporting these two operations at a minimum, but you can use more as long as both source and target Adapter know about them:

- "upsert"
- "delete"

The operation value is set by the source Adapter (which knows why this record is being transmitted) and consumed by the target Adapter so that it can apply the appropriate action to the record in the target external system.

- If you don't specify an operation on your records, the default is "upsert".
- You can mix operations in the same list of records.

Basic Source Adapter Interfaces

These interfaces are your basic building blocks for source Adapters. Each allows your Apex class to be used in a different style of Link run.

- *SourceAdapterForPull* - Implement if your Adapter can fetch records from an external system. This is the most common source Adapter variant.
- *SourceAdapterForRawDataPush* - Implement if your Adapter is parsing raw data (ex: a JSON payload) as the beginning of a Link run. Useful for realtime Link runs where an external system is *dropping data off for Valence using the Apex REST API*.

Tip: Any Adapter that implements *SourceAdapterForPull* should also immediately implement *SourceAdapterScope-Serializer*. It's easy to implement and it allows Links that use your source Adapter to run in parallel mode, which is much, much faster.

Source Adapter Enhancement Interfaces

These interfaces build on top of the basic ones, and register your Adapter either for extra behavior it needs from Valence, or for extra functionality it can offer Valence.

- *ChainFetchAdapter* - Implement if your Adapter is fetching data from a system that cannot predict in advance how many records (or how many batches) will be needed to retrieve all the records. This interface allows you to alternate record fetches with record processing indefinitely until all source records are exhausted.
- *ConfigurableSourceAdapter* - Implement if your source Adapter is user-configurable (see *Configurability*).
- *DelayedPlanningAdapter* - Implement if your Adapter needs a bit of real-world time between when it is asked for data and when it is ready to serve that data.

Being a Data Target

If you are a target Adapter, your basic responsibility is to write *RecordInFlight* instances to an external system, and mark them up with the results of that operation.

A RecordInFlight is really just a fancy Map of key-value pairs. You can extract the `Map<String, Object>` that was constructed for you to deliver like this:

```
Map<String, Object> oneRecord = aRecordInFlight.getProperties();
```

The collection of RecordInFlight instances you are handed is a live collection, and any changes you make to them will be conveyed to Valence. This means as you process the collection and try to write it to your adapter system, you should link those results back to each individual RecordInFlight so you can tell Valence how it went.

Listing 19: Example of linking API response to each record

```
List<Map<String,Object>> rawData = new List<Map<String, Object>>();
for(valence.RecordInFlight record : records) {
        rawData.add(record.getProperties());
}

List<Result> results = sendToAPI(rawData);

for(Integer i = 0, j = results.size(); i < j; i++) {
```

(continues on next page)

(continued from previous page)

```
        records[i].setCreated(results[i].createdNewRecord);

        records[i].setSuccess(results[i].success);

        if(results[i].success == false) {
                records[i].addError(results[i].errorMessage);
        }
}

private class Result {
        private Boolean success;
        private Boolean createdNewRecord;
        private String errorMessage;
}
```

Operation

Adding on to our discussion of *Record Operation* from earlier, it is the target Adapter's responsibility to apply the correct action that a record needs.

Don't assume all records you receive are upserts, some might be deletes. Typically you'll end up sorting RecordInFlight instances into different buckets based on their **operation** and resolve them separately.

If you don't support deleting records, mark those records in some way (probably with something like `record.ignore('Delete operation not supported.')`).

Not all admins want to allow records to be deleted, so you may want to add a configuration option to your Adapter to let an admin toggle deletes on or off.

Basic Target Adapter Interfaces

Target Adapters are a bit simpler than source Adapters in that there is only one basic interface:

- *TargetAdapter* - Implement if your Adapter can be the endpoint of a Link, i.e. the place where records are sent at the end of processing.

Warning: It is a **requirement** that your target Adapter respects the Link setting for *Testing Mode*, which an admin user sets when they don't want records to actually persist into the target system.

You will know if a Link run is in testing mode if the *LinkContext* `testingMode` boolean property is set to true. At a minimum, you can simply immediately return from pushRecords():

```
public void pushRecords(valence.LinkContext context, List<valence.RecordInFlight>␣
↪records) {

        // don't send any data if this Link is running in testing mode
        if(context.testingMode == true) {
                return;
        }
```

If possible, it's nice if you have a mechanism to do a trial push where you can test a write but roll back so that you can attach any errors or warnings to the RecordInFlight instances. Very few APIs support a mechanic like this, so we don't expect it but it's a nice to have.

Target Adapter Enhancement Interfaces

- *ConfigurableTargetAdapter* - Implement if your target Adapter is user-configurable (see *Configurability*).

5.24.5 Handling Errors

Every Adapter may potentially run into an exception or error when working with records.

If the issue is isolated to a single record and you can proceed, flag that record:

```
try {
        doSomethingWithARecord(record);
} catch(Exception e) {
        record.addError('Failed to flim flam the jibber jabber', e);
}
```

If the issue is catastrophic and your Adapter cannot continue, throw a `valence.AdapterException`:

```
throw new valence.AdapterException('Things have gone very badly wrong.');
```

If you have a causing Exception, you can wrap it:

```
throw new valence.AdapterException('Things have gone very badly wrong.',␣
↪exceptionThatCausedIt);
```

5.24.6 Test Coverage

When you're ready to start writing test coverage for your Adapter, check out *Writing Test Coverage*.

5.24.7 Summary

Adapters are the lifeblood of Valence, and as you can see there's a huge amount of tooling and support to help them shine. There's very few edge cases or situations where there is not already a feature or interface that will help you accomplish your goals.

Don't forget to take a peek at our open-source Adapters to see if what you need already exists, or to get inspiration and guidance on how to build them.

5.25 Adapter Authentication

5.25.1 Named Credentials

Overview

Adapters that reach out to *External Systems* to fetch or drop off data will need to authenticate themselves with that system.

Valence uses the native Salesforce Named Credential feature to handle:

1. Defining URLs to reach external systems to interact with
2. Configuring authentication details and storing identity secrets

Named Credentials are a fantastic feature that allows a Salesforce admin to define an endpoint (URL) and authentication information to access that endpoint.

They are a nice abstraction layer and we've taken advantage of that so a user can easily swap between specific instances of external systems without needing to change any code.

To allow Valence to hand you Named Credentials to work with, your Adapter will need to implement *NamedCredentialAdapter*.

Note: There are two booleans in Adapter registration that you will want to make decisions about:

- RequiresNamedCredentialForSchema__c - true if you need credentials to discover schema (some Adapters have their schema baked in, or in a flat file somewhere)
- RequiresNamedCredentialForData__c - certainly true if you're implementing NamedCredentialAdapter

These booleans drive whether Valence calls setNamedCredential() before it calls things like getTables() or fetchRecords().

Named Credential Settings

Named Credentials have some settings for headers that you should be familiar with.

- Generate Authorization Header: leave this checked if you are using a totally vanilla authentication type
- Allow Merge Fields in HTTP Header: check this (and uncheck the others) if you are building your own header values with credential information
- Allow Merge Fields in HTTP Body: check this (and uncheck the others) if the API you are talking to requires that you send user credentials as part of the message body

Supported Authentication Types

Out of the box Named Credentials support these types of authentication:

1. Standard username + password header authentication
2. OAuth using the "authorization code" grant type (but not any of the other types)
3. AWS Signature Version 4
4. JWT

5. JWT Token Exchange

Creative Authentication Types

You can actually get creative and squeeze a few more options out of these existing ones by using using the user + password option but turning off the normal authentication header.

For example, some APIs expect an “API token” or “API key” as a header value. You can still used Named Credentials for this! Just put a dummy value in the username field, the token in the password field, and then in your class you can do something like:

```
request.setHeader('Api-Token', '{!$Credential.Password}');
```

Apex callouts let you use special merge fields that will pull in values from the Named Credential without forcing you to store them, hardcode them, or even look at them.

Warning: For the above technique, be sure to check the option to “Allow Merge Fields in HTTP Header”, and uncheck “Generate Authorization Header” (since you’re rolling your own!).

Here’s another example of going a little outside the box with Named Credentials: the Intacct API expects a very specific format where the authentication details are part of the message payload. To satisfy this we set the username and password in a Named Credential, uncheck “Generate Authorization Header”, check “Allow Merge Fields in HTTP Body”, and then we can bring in the username and password using merge fields.

Finally, you can also get creative with the OAuth flow stuff because Salesforce provides some ways to create a custom Authentication Provider using a custom metadata type and an Apex class.

Our CEO actually did a video course on all these techniques called Authenticating External App and Service Integrations with Salesforce on the Pluralsight website, if you’d like a deeper dive and live examples. Of particular interest will be the sections **Calling out from Salesforce: Out of the Box** and **Calling out from Salesforce: Custom Authentication Providers**.

5.25.2 Additional Auth Info

Sometimes an API’s authentication requires an extra piece of information beyond user credentials, such as a “realm id” or “database name” or similar.

The best place at the moment to put that sort of information is to make your Adapter configurable and have the User set the value when they are configuring a Link.

5.26 Additional Table Details

It’s not uncommon for a running Adapter to need to know more about a source table or target table than just its name. Maybe the URL to call can’t be derived from the table name, or perhaps you need to know whether it should be HTTP POST or HTTP PATCH when writing to that table. Or maybe you need to remember which database the table is from.

There is typically a timing problem here: your Adapter knows all about the table when it is doing schema work to return a value from **getTables()**, but at runtime all you have to go on is the table name passed to you as part of the *LinkContext*.

Regardless of what pieces of information your Adapter needs, you can use the “table details” field to track them. Much like the way scopes behave during record fetching, table details are basically a mailbox your adapter can put things into for later retrieval at runtime.

5.26.1 Putting Values In

You store additional details about a table as part of building your response to **getTables()** (from *SchemaAdapter*). What you put into the field is up to you. Leave it empty, use a single value, or put some kind of data structure in.

```
Table.create('Company').withLabel('Company').withDescription('Definitions of businesses
↪').withTableDetails('/url/to/this/resource').build();
```

Here's a more complex example:

```
class TableInfo {
        String database;
        String format;
}
TableInfo info = TableInfo();
info.database = 'marketing';
info.format = 'XML';
Table.create('Company').withLabel('Company').withDescription('Definitions of businesses
↪').withTableDetails(JSON.serialize(info)).build();
```

5.26.2 Taking Values Out

You retrieve the value or values at runtime by inspecting either the **linkSourceTableDetails** or the **linkTargetTableDetails** properties on your *LinkContext*.

```
String urlToCall = linkContext.linkSourceTableDetails; // expecting the details to hold␣
↪the URL
```

And our more complex example:

```
class TableInfo {
        String database;
        String format;
}
TableInfo info = (TableInfo)JSON.deserialize(linkContext.linkTargetTableDetails,␣
↪TableInfo.class);
```

5.27 Creating A Filter

We're going to walk through how to tackle building a Valence Filter. What to think about, where to start, what's important.

If you haven't already read the primer on how Valence works overall, definitely *read that first*. Also make sure you have read *Extensions* and *Configurability*.

Ok, up to speed? Let's get to work.

5.27.1 The Beginning

A Filter's job is to take a peek at records as they are traveling from a data source to a data target. The Filter might add, change, or remove some values. A Filter might actually *filter* and decide that some records should not move on to delivery. There are many variations of what Filters do. If you want to take a peek at some existing Filters as you read this article, you can find them on GitHub.

Any Valence extension Apex class is going to implement various Valence-provided interfaces in order to be a cog in the Valence machine. These interfaces are like a contract: they define what Valence will provide to you, and also what Valence expects from you. It's always good to start small, and implement just one or two interfaces then build up from there.

Unless you're building a *LinkSplitFilter* for *Link Splits*, your Filter will implement *TransformationFilter*. This is our base Filter interface and is very simple:

```
global interface TransformationFilter {

        Boolean validFor(LinkContext context);

        void process(LinkContext context, List<RecordInFlight> records);
}
```

The first method, `validFor()` is asking our Filter if it would be appropriate to use for the Link described by the *LinkContext*. Most Filters return true from this method most of the time, but it's valuable to have it to cover the exceptions. For example, the Relations Filter that comes with Valence helps to populate Lookup and Master-Detail fields, so that Filter doesn't make sense to use on a Link going from Salesforce outbound to an external system. In this example, that Filter returns "false" if the target of the Link is not the local Salesforce org. Returning false means Valence will not show your Filter to users as an option when they are configuring that particular Link.

The second method is a Filter's bread and butter. We are handed a context and some *RecordInFlight* instances and asked to interact with them. This method has a return type of `void` because the records parameter we are passed is a live collection. If we make changes or mark up these records, the next Filter in line will see those changes, and so on down the line ending with the target Adapter that will deliver the records.

Filter Order

Filters are selected and arranged in an order by the admin user when configuring a Link, and the selected Filters fire sequentially in that order during a Link run. Each Filter will receive the work product of the previous Filter(s), which means you can put together some sophisticated behaviors where Filters work together to combine functionality.

There is one special Filter that is required to be included for all Links called the **Mappings Filter**. Most of selecting Filter order is choosing which Filters will run before the Mappings Filter and which ones will run afterwards.

Incidentally, even Valence Filters that come packaged with the app implement the same interfaces you do and abide by the same behaviors. The Mappings Filter implements TransformationFilter and does its magic in its `process()` method!

5.27.2 Working with RecordInFlight

You'll want to become very familiar with the methods available on each *RecordInFlight* instance so that you understand what sort of manipulations are available to you.

The first thing to understand is that a record has two internal maps:

1. **originalProperties** - what the record looked like when it was received from the data source
2. **properties** - what the record will look like when it is written to the target system

Typically Filters are modifying the **properties** map, but some Filters actually manipulate the **originalProperties** map. The one you work with will depend. Sometimes its based on whether your Filter is more familiar with the source system (and its field names) or the target system (and its field names). Other times its based on what other Filters you are trying to influence or support.

Here's a scenario that will help clarify this idea of Filter collaboration:

> The Mappings Filter is responsible for taking values in the **originalProperties** map and putting them into the **properties** map. Basically, any mapping that has been defined has a source field that exists in the **originalProperties** map, and Mappings will take that value and write it into the mapping target field in **properties**. So if your Filter runs after Mappings already ran, and modifies **originalProperties**, nothing is going to happen because we've already moved on to working with **properties**. However, if your Filter makes an adjustment to **originalProperties** *before* Mappings runs, then you can put stuff in front of Mappings for it to vacuum up and do its thing with.
>
> This is exactly how the Constant Filter works. Admin users need to make sure it runs before Mappings in their ordered Filter list. It creates fields and puts them into **originalProperties**, and then users define mappings with those fields, and the Mappings Filter actually moves the constant value from the source field to whatever target field the user has chosen in their mapping.
>
> So, in summary, the Constant Filter is complementing and enhancing something another Filter already does (in this case the Mappings Filter).

Separate from working with data, there are some other useful methods on RecordInFlight that Filters often invoke:

- addError() - attach an error to this record, which will block it from being delivered and also surface the error in the user interface for admins to troubleshoot
- addWarning() - attach a warning to this record, which does not block delivery, but will also show up in the interface for admins to see
- ignore() - stop a record from being delivered, but semantically this isn't so much an error as it is simply a record you know we're not interested in delivering (also shows up in interface for admins to see…detecting a pattern?)

5.27.3 Participating in Schema Discovery

Valence has a *rich understanding of schema*, and as part of our obsession with giving admins great context about what they're looking at, we want to make sure that the schema of their records that they see reflects changes due to the transformations and configurations they set up.

Since Filters can have a big impact on records and what they look like, it's important that your Filter be able to describe what it does in a programmatic way. This is done by implementing *SchemaAwareTransformationFilter*.

The interface article has a really clear breakdown of how to properly inform Valence about your impact on schema.

5.27.4 Configuring Your Filter

There are two flavors of how your Filter might be configurable that are described below.

You will also want to read *Configurability* to learn how configurations are consumed by your Apex class, and how to surface a user interface for admins to do the actual configuring.

Configurable Per Link

/filter-interfaces/configurable-per-link is for Filters that have admin-configured behavior but that are not connected to a specific mapping. These are Filters that affect the record holistically, and an admin user can set up one or more configurations on the same Link for the same Filter.

Examples:

- The Constant Filter injects one new source field for each configuration that an admin creates, and that source fields holds some static value
- The Object Builder Filter allows an admin to compose a Map from existing source fields that will become a single source field value

---

Note: Not every Filter has to be configurable to be useful. For example, there is an SF Groomer Filter that comes packaged with Valence that looks at every field that will be written to Salesforce and attempts to tweak its value to conform with the field type being written to (so things like "true" and "false" will be converted to boolean true and boolean false when writing to a checkbox field).

Configurable Per Mapping

Many Filters apply a transformation to or interpret the value of a specific field selected by the admin user. For this pattern the *ConfigurablePerMappingFilter* interface is perfect.

Implementing this interface allows admins to attach a configuration for your Filter to each *Mapping* they want to apply its logic to.

Examples:

- Our guide on *building a Cutoff Date Filter* walks you through setting up a Filter where an admin attaches it to one or more date fields, and the configuration is selecting a date literal to use as a threshold. The Filter evaluates the value in the date field it is attached to, and if that date is before the configured cutoff that record is ignored.
- The **Relationships Filter** that comes packaged with Valence is attached to mappings that hold unique identifiers from external systems, and walks admins through how to use that identifier to populate a Master-Detail or Lookup field in Salesforce.

5.27.5 Handling Errors

Every Filter may potentially run into an exception or error when working with records.

If the issue is isolated to a single record and you can proceed, flag that record:

```
try {
	manipulateRecord(record);
} catch(Exception e) {
```

(continues on next page)

(continued from previous page)

```
        record.addError('Failed to flim flam the jibber jabber', e);
}
```

If the issue is catastrophic and your Filter cannot continue, throw a `valence.FilterException`:

```
throw new valence.FilterException('Things have gone very badly wrong.');
```

If you have a causing Exception, you can wrap it:

```
throw new valence.FilterException('Things have gone very badly wrong.',␣
↪exceptionThatCausedIt);
```

5.27.6 Test Coverage

When you're ready to start writing test coverage for your Filter, check out *Writing Test Coverage*.

5.27.7 Summary

There are a million and one different ways people transform records in their integrations. Rather than try to guess every possible combination and bake them into the engine, we've designed and exposed a robust transformation framework that we use internally and is also available for you.

Have a look through our open-source Filters to see if what you need already exists, or to get inspiration and guidance on how to build them.

Start small with *TransformationFilter*, and then layer in schema and configurability to make your Filter a sophisticated participant in the fabric of Valence!

5.28 AdapterException

Special **Exception** class that we encourage you to use in your Adapters to indicate that something has gone wrong that cannot be recovered from.

Using this custom exception allows us to surface the messages you write on the exception directly to users to help them understand what went wrong.

5.28.1 Definition

```
/**
 * Exception related to, and thrown by, Adapters.
 */
global class AdapterException extends Exception {}
```

5.28.2 Usage

You can throw the exception outright if you know things are in a bad state.

Example 1

```
if(thingsAreBroken) {
    throw new valence.AdapterException('Everything exploded!');
}
```

Example 2

Or you can wrap a low-level exception and add some context, giving us the best chance of surfacing an accurate error.

```
try {
    complexOperation();
}
catch(NullPointerException npe) {
    throw new valence.AdapterException('Complex Operation failed.', npe);
}
```

5.29 CSVReader

CSVReader makes it easy to parse out raw strings that contain CSV-formatted data. CSV parsing can be deceptively difficult once you start getting into edge cases around commas, quotes, line breaks, etc, so this is a handy class to have around when you're reading CSV data from *External Systems*.

The source code for this class is an adaptation of an open-source library from Marty Chang.

5.29.1 Definition

```
global static List<List<String>> readCSVData(String rawData); // defaults to using \n as␣
↪the line ending character

global static List<List<String>> readCSVData(String rawData, String lineEnding);
```

5.29.2 Example Usage

```
// parse the response
List<List<String>> rows = valence.CSVReader.readCSVData(apiResponseBody);

// extract headers from first line of CSV data
List<String> headers = rows[0];

List<valence.RecordInFlight> records = new List<valence.RecordInFlight>();

// iterate over response records and create RecordInFlight instances from them
```

(continues on next page)

(continued from previous page)

```
for(Integer i = 1, j = rows.size(); i < j; i++) { // deliberately start with i = 1 to␣
↪skip the header row

        Map<String, Object> properties = new Map<String, Object>();

        for(Integer k = 0, l = headers.size(); k < l; k++) {
                properties.put(headers[k], rows[i][k]);
        }

        records.add(new valence.RecordInFlight(properties));
}
```

5.30 FetchStrategy

FetchStrategy allows your source *Adapter* to tell Valence how best to ask your Adapter for records. It is an implementation of the Strategy Pattern, and is used by Adapters that implement the *SourceAdapterForPull* interface.

This helper class is a critical part of how your Adapter and Valence work together when fetching data. In a nutshell:

1. Valence asks your Adapter for a FetchStrategy to use to get the data
2. Your Adapter tells Valence which one should be used
3. Valence proceeds to set up a Link run and interact with your Adapter according to the FetchStrategy you asked for

There are a few different "strategies" available for your selection. You can even mix and match them depending on what's going on, perhaps using **IMMEDIATE** if there are only a few records, or **SCOPES** if you have many thousand to retrieve.

Tip: Every strategy (except NO_RECORDS) has an alternate factory method that accepts an extra parameter value called "expectedTotalRecords". If during planning you know exactly how many records you intend to fetch, always use this version of the method. This helps users understand how many records are going to be fetched during a Link run. If you don't know how many records you'll be fetching ahead of time, that's perfectly fine, just use the simpler version of the method.

5.30.1 Strategy: Immediate

The **IMMEDIATE** strategy calls fetchRecords() immediately in the same execution context (any state in your Adapter is still there). A null value is passed as the scope parameter.

This is the simplest and easiest strategy to work with. You can get some decent mileage out of it before you have to move to **SCOPES**.

One nice thing is that Valence abstracts away some of the Salesforce limits, so for example you can return more than 10,000 records from fetchRecords() (which would normally hit the DML row limit) and that's not a problem.

You do, however, still have to stay under the heap size limit of 12 MB. If you think you might hit this limit, consider using **SCOPES** and breaking your result set down across multiple execution contexts.

Definition

```
global static FetchStrategy immediate();

global static FetchStrategy immediate(Long expectedTotalRecords);
```

Example Usage

```
public valence.FetchStrategy planFetch(valence.LinkContext context) {
	return valence.FetchStrategy.immediate();
}

public List<valence.RecordInFlight> fetchRecords(valence.LinkContext context, Object␣
↪scope) {

	// retrieve some records
	List<valence.RecordInFlight> records = goGetRecords();

	return records;
}
```

5.30.2 Strategy: Scopes

Warning: Reminder: unlike **IMMEDIATE**, when fetchRecords() is called on your Adapter it happens in an entirely new execution context. This means your Adapter state is *totally wiped clean* between calls! If you need something, put it into your scopes variable during planning.

The **SCOPES** strategy breaks up fetching records from the external system into multiple Salesforce execution contexts (one per scope). Each scope will have a fresh set of limits and state. In order to use this strategy, during your planFetch() call you need to figure out how many scopes are necessary. Compare **context.batchSizeLimit** to whatever hard limits your external system has, and use the lower of the two as your batch size.

During planning your goal is to build a list of scopes that give you whatever details you need to be able to retrieve each batch of records from the external system. Maybe that's a unique identifier for the job combined with an offset value or page number. It varies but the scope shape is entirely up to you.

This is the most common FetchStrategy used in production Valence instances.

Tip: If you are working with an API that cannot tell you upfront how many total records you will be fetching, you have two options. You can "chain" executions using the *ChainFetchAdapter*, or you can scroll down and look at **CUMULATIVE_SCOPES**.

Definition

```
global static FetchStrategy scopes(List<Object> scopes);

global static FetchStrategy scopes(List<Object> scopes, Long expectedTotalRecords);
```

Example Usage

```
public valence.FetchStrategy planFetch(valence.LinkContext context) {

    String requestId = getRequest(); // some method to get whatever info you need about␣
↪the request
    Integer total = countExternalRecords(); // and grab a record count, for example

    // determine how many records you can fetch at a time
    Integer batchSize = context.batchSizeLimit < EXTERNAL_LIMIT ? context.batchSizeLimit␣
↪: EXTERNAL_LIMIT;

    // build our list of custom scopes
    List<MyScope> scopes = new List<MyScope>();
    Integer offset = 0;
    while(offset < total) {
        scopes.add(new MyScope(requestId, offset));
        offset += batchSize;
    }

    // tell Valence we're using the SCOPES strategy
    return valence.FetchStrategy.scopes(scopes, total);
}

public List<valence.RecordInFlight> fetchRecords(valence.LinkContext context, Object␣
↪scope) {

    // cast to our custom scope class
    MyScope currentScope = (MyScope)scope;

    // retrieve some records from the external server with an offset, for example
    return fetchRecordsFromServer(currentScope.requestId, currentScope.offset);
}

public class MyScope {
    private String requestId;
    private Integer offset;

    public MyScope(String requestId, Integer offset) {
```

(continues on next page)

(continued from previous page)

```
        this.requestId = requestId;
        this.offset = offset;
    }
}
```

5.30.3 Strategy: Delay

The **DELAY** strategy allows you to pause for an arbitrary amount of time when starting up a Pull Link run. This is a very situational strategy but can be helpful for circumstances such as waiting for a file to be generated on an external server. Let's say you call into the server during planFetch() and describe the records you want. The external server generates a CSV file somewhere and then exposes it to you. You could use the **DELAY** strategy to wait until that CSV file is ready to be read, check if the file is ready, and if it is you'd read the generated file with your fetchRecords() call.

You can delay as many times as you need to. You can leave the duration of the delay up to Valence (15 seconds ~ 90 seconds), or you can specify a number of minutes to wait as a minimum, max of 10 minutes (handy if you know for sure that it'll be a few minutes before things are ready). If you need more than 10 minutes or if your resource isn't quite ready you can keep returning **DELAY** from planFetchAgain() for more time.

If you use the **DELAY** strategy your Adapter must also implement the *DelayedPlanningAdapter* interface. This interface adds an additional method that Valence will call after the delay is over. You are welcome to return **DELAY** again from this method call, and keep doing that over and over until you are ready to move on to fetching records.

Definition

```
global static FetchStrategy delay(Integer minutes, Object scope);

global static FetchStrategy delay(Integer minutes, Object scope, Long␣
↪expectedTotalRecords);
```

Example Usage

```
private String filePath = null;

public valence.FetchStrategy planFetch(valence.LinkContext context) {
        // do some asynchronous operation that you know will take 3-5 minutes
        String fileId = generateFile();

        // ask Valence to call you back in 10 minutes
        return valence.FetchStrategy.delay(10, fileId);
}

public valence.FetchStrategy planFetchAgain(valence.LinkContext context, Object scope) {

        // get the state that we previously stashed in the scope object
        String previousFileId = (String)scope;

        // check to see if that file is ready yet
        Boolean ready = checkFileStatus(previousFileId);
```

(continues on next page)

(continued from previous page)

```
        if(ready) {
                filePath = getFilePath(previousFileId);
                return valence.FetchStrategy.immediate(); // will call fetchRecords() in␣
↪this same execution context with a null second parameter, which is why we set filePath
        } else {
                // wait two more minutes then try again
                return valence.FetchStrategy.delay(2, previousFileId);
        }
}

public List<valence.RecordInFlight> fetchRecords(valence.LinkContext context, Object␣
↪scope) {

        // retrieve and parse the file
        List<valence.RecordInFlight> records = retrieveAndParse(filePath);

        return records;
}
```

Tip: If you don't want to specify a certain number of minutes to wait and instead let Valence decide, pass *null* as the first parameter: return valence.FetchStrategy.delay(null, myScope);

5.30.4 Strategy: Cumulative Scopes

The **CUMULATIVE_SCOPES** strategy is like a mashup of **SCOPES** and **DELAY**. It allows you to give a partial list of scopes to Valence, then wait a little bit and then give Valence another partial list, repeating as needed until you are satisfied you have collected all the scopes you need.

Why is this helpful?

This is definitely an edge-case FetchStrategy, but for those edge cases it's just the tool for the job. Example: imagine an API that can either return lists of record IDs, or a single full record. A normal **SCOPES** strategy expects you to know exactly how many scopes you are going to need up front, and for each one to be defined and returned to Valence in a single collection. What if this hypothetical API had millions of records? You can paginate through the ID list but that's not going to give you full records.

So what do you do?

In this example you could use CUMULATIVE_SCOPES to essentially do a two-dimensional record fetch. Turn each record ID into a single scope that will be used during the real Link run to fetch a full record, then paginate through IDs one page at a time to build up your record scopes. Maybe you get 1000 IDs in a page, give Valence 1000 scopes, and then let Valence call you back to get the next page of IDs. Repeat as long as needed.

Any Adapter that returns CUMULATIVE_SCOPES as a FetchStrategy must also implement *DelayedPlanningAdapter* and *SourceAdapterScopeSerializer*. DelayedPlanningAdapter is used to do a planFetchAgain() when you are building up scopes, and SourceAdapterScopeSerializer gives you control over how your scope instances are serialized by Valence. Carefully read about each of these interfaces.

Note: If this Link has its setting for "enableParallelProcessing" set to "true", scopes will start being processed as soon as you return your first collection of scopes and continue to be processed as you return each set until you are collecting scopes. If "false", no scopes are processed until the final collection of scopes is given to Valence.

Note: When you are responding with what you know will be your last collection of scopes, return null for the 'Object scope' parameter so Valence knows you're done. This is a little different to how the **DELAY** FetchStrategy works.

Tip: It's fine to use the expectedTotalRecords version on one of your response (typically the first) and then the shorter version on your others. We'll remember the number.

Definition

```
global static FetchStrategy cumulativeScopes(List<Object> scopes, Integer minutes,␣
↪Object scope);

global static FetchStrategy cumulativeScopes(List<Object> scopes, Integer minutes,␣
↪Object scope, Long expectedTotalRecords);
```

Example Usage

```
public valence.FetchStrategy planFetch(valence.LinkContext context) {

        Long total = countExternalRecords(); // grab a record count, for example

        // fetch a page of records from your external server and store it in some kind␣
↪of response object you can work with
        ResponseObject response = fetchPageOfRecordIds(1);

        // build our list of custom scopes for fetching individual full records
        List<FetchFullRecordScope> scopes = new List<FetchFullRecordScope>();
        for(String recordId : response.identifiersOnThisPage) {
                scopes.add(new FetchFullRecordScope(recordId));
        }

        FetchPageOfIDsScope nextPageScope = null;
        if(response.hasMoreRecords == true) {
                nextPageScope = new FetchPageOfIDsScope(2);
        }

        // tell Valence we're using the CUMULATIVE_SCOPES strategy (but in this example␣
↪we don't need an artificial delay, so passing null for 'minutes' parameter)
        return valence.FetchStrategy.cumulativeScopes(scopes, null, nextPageScope,␣
↪total);
}

public valence.FetchStrategy planFetchAgain(valence.LinkContext context, Object scope) {

        // get the state that we previously stashed in the scope object
        FetchPageOfIDsScope currentPageScope = (FetchPageOfIDsScope)scope;

        // fetch a page of records from your external server and store it in some kind␣
```

(continues on next page)

(continued from previous page)

```
↪of response object you can work with
        ResponseObject response = fetchPageOfRecordIds(currentPageScope.pageNumber);

        // build another list of full record scopes using this fresh page of identifiers
        List<FetchFullRecordScope> scopes = new List<FetchFullRecordScope>();
        for(String recordId : response.identifiersOnThisPage) {
                scopes.add(new FetchFullRecordScope(recordId));
        }

        FetchPageOfIDsScope nextPageScope = null;
        if(response.hasMoreRecords == true) {
                nextPageScope = new FetchPageOfIDsScope(currentPageScope.pageNumber + 1);
        }

        return valence.FetchStrategy.cumulativeScopes(scopes, null, nextPageScope);
}

public List<valence.RecordInFlight> fetchRecords(valence.LinkContext context, Object␣
↪scope) {

        // cast to our custom scope class
        FetchFullRecordScope currentScope = (FetchFullRecordScope)scope;

        // retrieve this record from the external server
        return fetchRecordFromServer(currentScope.recordIdentifier);
}

public class FetchPageOfIDsScope {

        private Integer pageNumber;

        public FetchPageOfIDsScope(Integer pageNumber) {
                this.pageNumber = pageNumber;
        }
}

public class FetchFullRecordScope {

        private String recordIdentifier;

        public FetchFullRecordScope(String recordIdentifier) {
                this.recordIdentifier = recordIdentifier;
        }
}
```

5.30.5 Strategy: Locator

It's unlikely you will ever need to use the **LOCATOR** strategy. This is a specialized strategy that uses a Database.QueryLocator to iterate over large quantities (millions) of records inside the local Salesforce org. Normally you are going to leverage the built-in Local Salesforce Adapter that comes with Valence for extracting records from the local Salesforce org. However, this strategy is available to you should you need to do this kind of thing yourself.

If you use the **LOCATOR** strategy your Adapter must also implement the *SourceAdapterForSObjectPush* interface. Valence will retrieve records using the locator and feed them to your sObject push method to process them. Your Adapter's fetchRecords() is never called.

Definition

```
global static FetchStrategy queryLocator(String query);

global static FetchStrategy queryLocator(String query, Long expectedTotalRecords);
```

Example Usage

```
public valence.FetchStrategy planFetch(valence.LinkContext context) {
        return valence.FetchStrategy.locator('SELECT Id, Name, Phone FROM Account');
}

public List<valence.RecordInFlight> buildRecords(valence.LinkContext context, List
↪<sObject> records) {
        // process sObject records in batches of context.batchSizeLimit or 2,000,␣
↪whichever is smaller
}
```

5.30.6 Strategy: No Records

Use the **NO_RECORDS** strategy if during planning you notice that you have no records that you need to fetch.

This will short-circuit the rest of processing; your fetchRecords() method will not be invoked, and the SyncEvent is immediately closed.

Definition

```
global static FetchStrategy noRecords();
```

Example Usage

```
public valence.FetchStrategy planFetch(valence.LinkContext context) {

        Integer count = countRecordsToFetch();

        return count > 0 ? valence.FetchStrategy.immediate() : valence.FetchStrategy.
↪noRecords();
}

public List<valence.RecordInFlight> fetchRecords(valence.LinkContext context, Object␣
↪scope) {

        // retrieve some records
        List<valence.RecordInFlight> records = goGetRecords();

        return records;
}
```

5.30.7 Test Coverage

There are a few instance methods in the FetchStrategy class to help you write test coverage against your Adapter's planFetch() method.

Definition

```
global String checkStrategyType();

global Long checkExpectedTotalRecords();

global Integer checkScopeCount();
```

Example Usage

```
@IsTest
private static void testPlanFetchImmediate() {

        // set up a callout mock that simulates the external system telling us there are␣
↪17 records available to pick up
        Test.setMock(HttpCalloutMock.class, new MyAPIMockClass(MyAPIMockClass.Response.
↪COUNT_DIRTY_RECORDS_17));

        valence.LinkContext context = new valence.LinkContext();
        context.linkSourceName = 'SomeTable';
        context.batchSizeLimit = 20;

        Test.startTest();
        MyAdapterClass adapter = new MyAdapterClass();
        valence.FetchStrategy strategy = adapter.planFetch(context);
```

(continues on next page)

(continued from previous page)

```
        Test.stopTest();

        // we are expecting the IMMEDIATE strategy because all the outstanding records␣
↪would fit in one batch
        System.assertEquals('IMMEDIATE', strategy.checkStrategyType());
        System.assertEquals(17, strategy.checkExpectedTotalRecords());
}

@IsTest
private static void testPlanFetchScopes() {

        // set up a callout mock that simulates the external system telling us there are␣
↪26 records available to pick up
        Test.setMock(HttpCalloutMock.class, new MyAPIMockClass(MyAPIMockClass.Response.
↪COUNT_DIRTY_RECORDS_26));

        valence.LinkContext context = new valence.LinkContext();
        context.linkSourceName = 'SomeTable';
        context.batchSizeLimit = 20;

        Test.startTest();
        MyAdapterClass adapter = new MyAdapterClass();
        valence.FetchStrategy strategy = adapter.planFetch(context);
        Test.stopTest();

        // we are expecting the SCOPES strategy because our record count exceeds our␣
↪batchSizeLimit so we'll need multiple batches
        System.assertEquals('SCOPES', strategy.checkStrategyType());
        System.assertEquals(26, strategy.checkExpectedTotalRecords());
        System.assertEquals(2, strategy.checkScopeCount());
}
```

5.31 Field

The Field class represents a property that a record may have. It is analogous to a table column, or—in Salesforce—an object field.

We refer to Fields when we are inspecting the Schema, or shape, of records for a particular table or object. You will encounter this Valence class if you are implementing the SchemaAdapter interface.

Fields are constructed using a builder pattern that uses a fluent interface to make it simple to construct Field instances with varying degrees of complexity.

You get a FieldBuilder by calling Field.create() and passing the API name of the Field you are constructing. This is the only required property, but it's an important one. This string value is expected to match exactly to what is used on the records that will be retrieved from, and given to, your Adapter.

5.31.1 Simple Field Example

```
valence.Field firstNameField = valence.Field.create('firstName').withLabel('First Name').
↪build();
```

5.31.2 More Complex Field Example

```
valence.Field startDateField = valence.Field.create('startDate')
    .withLabel('Start Date')
    .withDescription('What day this subscription was first activated')
    .withType('string')
    .withFormat('yyyy-MM-dd')
    .withExampleValue('2017-01-24')
    .build();
```

5.31.3 Nested Data Shapes

Sometimes you will need to work with more complex schema involving collections and nested data structures.

For this you will take advantage of **setMap()**, **setList()**, and **addChild()**.

Let's look at the code for describing the following data shape:

```
{
        "Name" : "Acme",
        "Type" : "Financial",
        "Website" : "acme.com",
        "Employees" : [
                {
                        "FirstName" : "Julie",
                        "LastName" : "Smith"
                },
                {
                        "FirstName" : "Thomas",
                        "LastName" : "Lincoln"
                }
        ],
        "Address" : {
                "Street" : "123 Main St",
                "City" : "Urbanville"
        }
}
```

```
Field.create('Employees').withDescription('Workers at this company').setList(true, false)
    .addChild(Field.create('FirstName').withLabel('First Name').withDescription('First␣
↪name of the person').withExampleValue('Julie').withType('String').build())
    .addChild(Field.create('LastName').withLabel('Last Name').withDescription('Last name␣
↪of the person').withExampleValue('Smith').withType('String').build())
    .build()
```

```
{
        "Name" : "Acme",
        "Type" : "Financial",
        "Website" : "acme.com",
        "Employees" : [
                {
                        "FirstName" : "Julie",
                        "LastName" : "Smith"
                },
                {
                        "FirstName" : "Thomas",
                        "LastName" : "Lincoln"
                }
        ],
        "Address" : {
                "Street" : "123 Main St",
                "City" : "Urbanville"
        }
}
```

```
Field.create('Address').setMap(true, false)
    .addChild(Field.create('Street').withExampleValue('123 Main St').withType('String').
↪build())
    .addChild(Field.create('City').withExampleValue('Urbanville').withType('String').
↪build())
    .build()
```

Tip: **setMap()** and **setList()** take an additional parameter to indicate if the field is "lazy". A lazy field has more structure beneath it, but you are not adding it right now with **addChild()**. This is intended to be paired with *LazyLoadSchemaAdapter* so Valence users can drill down into your data structure as they need do.

5.31.4 Properties

Data Type	Class Property	Builder Method	Description
String	name		*Required*. Expected to match actual record field names.
String	label	withLabel(String label)	User-friendly label for this field.
String	description	withDescription(String description)	Description of the field's purpose.
String	exampleValue	withExampleValue(String exampleValue)	An example value to help give a user context. Truncates to 20 characters.
String	defaultValue	withDefaultValue(String defaultValue)	The default value that will be used if this field is left blank. Truncates to 20 characters.
String	type	withType(String type)	The data type of the field. Truncates to 20 characters.
String	format	withFormat(String format)	The format of the value, if applicable. Truncates to 20 characters.
Boolean	isRequired	setRequired(Boolean isRequired)	True if this field must be present on every record.
Boolean	isCreateable	setCreateable(Boolean isCreateable)	True if this field can be set on record creation.
Boolean	isUpdateable	setUpdateable(Boolean isUpdateable)	True if this field can be set on record update.
Boolean	isEditable	setEditable(Boolean isEditable)	True if either isCreateable or isUpdateable have been set. Can also be set independently.
Boolean	isMap	setMap(Boolean isMap, Boolean isLazy)	True if this Field in the schema is a Map of other Fields (so, keyed)
Boolean	isList	setList(Boolean isList, Boolean isLazy)	True if this Field is a List of other Fields (no keys)
Boolean	isLazy		True if this List/Map is not going to have children set right now, but could have them lazily loaded later
		addChild(Field child)	Use this to nest fields under other fields in a hierarchy.

5.32 Field Path

A FieldPath is a sequence of *Field* instances that represent the path from the root of the schema down to a particular Field that is of interest.

If a Field instance is a "what", then a FieldPath is a "where". You can pretty easily think of a FieldPath as an array of strings where each item is the name of a field.

Tip: FieldPath is an appropriate tool when thinking about and interacting with schema information, but for actual record interactions we'll use *Property Path*.

5.32.1 Building an Instance

```
global static FieldPath buildPath(List<Field> fields);

global static FieldPath buildPath(Field field);

global static FieldPath buildPath(List<String> fieldNames);

global static FieldPath buildPath(String fieldName);
```

5.32.2 Instance Methods

```
// get the full chain of Fields
global List<Field> getSequence()

// get the Field at the end of the chain (the leaf)
global Field getLast()

// get the the field name of each Field in the sequence
global List<String> getSimplePath()
```

5.32.3 Example

Suppose you are working with the **LastName** field on the **Contact** object in Salesforce.

If we were just working within the **Contact** object, getSequence() would be a List with one Field instance:

```
// [LastName]

List<Field> fields = path.getSequence();
fields.size(); // 1
fields[0].name; // LastName
fields[0].type; // STRING
fields[0].isRequired; // true
fields[0].isList; // false
```

However, what if we were actually building a Link against the **Account** object, and wanted to refer to related **Contact** records? There is a Master-Detail field **Contact.AccountId**, whose inverse from the **Account** side is called **Contacts** (with an S).

So in this scenario, from the perspective of the **Account** object, we'd be interested in looking at **Contacts.LastName**.

```
// [Contacts,LastName]

List<Field> fields = path.getSequence();
fields.size(); // 2

fields[0].name; // Contacts
fields[0].type; // CHILDREN
fields[0].isRequired; // false
fields[0].isList; // true
```

(continues on next page)

(continued from previous page)

```
fields[1].name; // LastName
fields[1].type; // STRING
fields[1].isRequired; // true
fields[1].isList; // false
```

So the same Field (**Contact.LastName**) might be referenced with different paths. You might even have the same Field referenced by different paths in the same schema.

FieldPath helps us encapsulate this and makes it easier to work with arbitrarily-nested Fields.

Warning: As you can see from methods like buildPath(List<String> fieldNames), sometimes you will have a FieldPath whose Field instances have sparsely-populated properties (in this example, only Field.name would have a value); **name** and **isList** are the only two properties you can safely assume will always be set.

5.33 FilterException

Special Exception class that we encourage you to use in your Filters to indicate that something has gone wrong that cannot be recovered from.

Using this custom exception allows us to surface the messages you write on the exception directly to users to help them understand what went wrong.

5.33.1 Definition

```
/**
 * Exception related to, and thrown by, Filters.
 */
global class FilterException extends Exception {}
```

5.33.2 Usage

You can throw the exception outright if you know things are in a bad state.

Example 1

```
if(thingsAreBroken) {
    throw new valence.FilterException('Everything exploded!');
}
```

Example 2

Or you can wrap a low-level exception and add some context, giving us the best chance of surfacing an accurate error.

```
try {
    complexOperation();
}
catch(NullPointerException npe) {
    throw new valence.FilterException('Complex Operation failed.', npe);
}
```

5.34 JSONParse

JSONParse is a JSON parser we wrote in Apex and then open-sourced on Github. Reading JSON data in Apex is super annoying and we hope this parser makes you life a little better.

You don't need the Github class, this library is baked right into Valence. You can access it at **valence.JSONParse**.

5.34.1 Overview

Salesforce Apex JSON parser to make it easier to extract information from nested JSON structures.

If you're sick of writing code like this:

```
Map<String, Object> root = (Map<String, Object>)JSON.deserializeUntyped(someJSON);
Map<String, Object> menu = (Map<String, Object>)root.get('menu');
Map<String, Object> popup = (Map<String, Object>)menu.get('popup');
List<Object> menuitem = (List<Object>)menu.get('menuitem');
Map<String, Object> secondItem = (Map<String, Object>)menuitem.get(1);
String thingIActuallyWanted = String.valueOf(secondItem.get('name'));
```

. . . then this parser is for you! Voila!

```
String thingIActuallyWanted = new valence.JSONParse(someJSON).get('menu.popup.menuitem.
↪[1].name').getStringValue();
```

Do I have your attention? Great! Now let's go a little deeper.

5.34.2 Concepts

The idea of JSONParse is that a JSON payload is treated as a tree, with each node in the tree wrapped in an instance of JSONParse. So you start with a JSONParse instance at the root, and as you drill deeper into the nested data structure you are revealing yet more JSONParse instances.

At any time you can use your current JSONParse instance to get one of two collection types (Maps/Lists), or raw data primitives, depending on what this particular JSONParse node is wrapping (an object, an array, or a primitive).

A little fuzzy? That's OK, let's look at some examples.

5.34.3 Usage

Let's start with a simple example. Say we have the following JSON structure:

```
{"menu": {
  "id": "file",
  "value": "File",
  "popup": {
    "menuitem": [
      {"value": "New", "onclick": "CreateNewDoc()"},
      {"value": "Open", "onclick": "OpenDoc()"},
      {"value": "Close", "onclick": "CloseDoc()"}
    ]
  }
}}
```

We always start by instantiating JSONParse with a String value that holds some JSON:

```
valence.JSONParse root = new valence.JSONParse(someJSONData);
```

If we wanted to get to the `value` property inside `menu`, we would do this:

```
root.get('menu.value').getStringValue(); // "File"
```

But what is actually happening here? Let's be a little more verbose:

```
valence.JSONParse childNode = root.get('menu.value');
childNode.getStringValue(); // "File"
```

The `get()` method is our key workhorse method in JSONParse. It allows us to drill into the tree structure and always returns an instance of JSONParse.

For additional clarity:

```
System.debug(root.toStringPretty());
```

```
{
  "menu" : {
    "popup" : {
      "menuitem" : [ {
        "onclick" : "CreateNewDoc()",
        "value" : "New"
      }, {
        "onclick" : "OpenDoc()",
        "value" : "Open"
      }, {
        "onclick" : "CloseDoc()",
        "value" : "Close"
      } ]
    },
    "value" : "File",
    "id" : "file"
  }
}
```

```
System.debug(root.get('menu.popup').toStringPretty());
```

```
{
  "menuitem" : [ {
    "onclick" : "CreateNewDoc()",
    "value" : "New"
  }, {
    "onclick" : "OpenDoc()",
    "value" : "Open"
  }, {
    "onclick" : "CloseDoc()",
    "value" : "Close"
  } ]
}
```

```
System.debug(root.get('menu.popup.menuitem').toStringPretty());
```

```
[ {
  "onclick" : "CreateNewDoc()",
  "value" : "New"
}, {
  "onclick" : "OpenDoc()",
  "value" : "Open"
}, {
  "onclick" : "CloseDoc()",
  "value" : "Close"
} ]
```

```
System.debug(root.get('menu.popup.menuitem.[0]').toStringPretty());
```

```
{
  "onclick" : "CreateNewDoc()",
  "value" : "New"
}
```

You can see as we drill deeper and deeper into the data structure, we get smaller and smaller slices of the tree back.

Don't be misled by these examples that all start from `root`. You can just as easily drill partway down, do some stuff, then keep going. You can even do things like this:

```
root.get('menu.popup').get('menuitem').get('[2].onclick').getStringValue();
```

5.34.4 What can I pass to get()?

```
public valence.JSONParse get(String path) {}
```

The syntax for what you pass to the `get()` method is simple. You are passing a series of tokens separated by periods. A token is either:

1. An array token
2. A key token

Array tokens look like this: [2]

They are used to choose a specific item in an array.

The other token type, key tokens, are just simple strings that are going to be used to match on a JSON object property name. This matching is **case sensitive**.

You can mix and match these two token types to your heart's content. Just remember you are following your JSON data structure, so your get() path that you send should match it!

Tokens are always separated by a period. Here's a common mistake (don't do this):

```
// NOT VALID SYNTAX
root.get('menu.popup.menuitem[0]');

// VALID SYNTAX
root.get('menu.popup.menuitem.[0]');
```

5.34.5 Working with Collections

If you'd like to work with your collection nodes (object = Map, array = List), there are two methods on JSONParse:

```
public Map<String, valence.JSONParse> asMap() {}
public List<valence.JSONParse> asList() {}
```

So, to build on our previous examples, you could do something like this:

```
for(valence.JSONParse node : root.get('menu.popup.menuitem').asList()) {

    System.debug('Onclick: ' + node.get('onclick').toStringValue());
}
```

Or this:

```
Map<String, valence.JSONParse> menuProperties = root.get('menu').asMap();

System.debug(menuProperties.keySet()); // (id, value, popup)

for(String key : menuProperties.keySet()) {

  valence.JSONParse node = menuProperties.get(key);
}
```

You can of course nest and repeat these patterns, to drill into arbitrarily complex data structures.

5.34.6 Dynamic Inspection

There's some discovery built into the parser so you can explore the JSON structure without knowing the shape in advance.

To do so, simply combine the two collection methods you just saw with these utility methods:

```
public Boolean isObject() {}
public Boolean isArray() {}
```

These two methods peek under the covers at the wrapped data and give you some information about what's inside. Here's an arbitrary example, where I use recursion to perform a dynamic inspection of the entire JSON tree. Obviously this is contrived but it should give you an idea of what's possible!

You can copy this entire snippet into Anonymous Apex and run it yourself.

Anonymous Apex Snippet

```
public void explore(valence.JSONParse node, Integer depth) {

    if(!(node.isObject() || node.isArray())) {
        System.debug( '*'.repeat(depth) + node.getValue());
    }

    if(node.isObject()) {
        for(String key : node.asMap().keySet()) {
            explore(node.get(key), depth + 1);
        }
    }

    if(node.isArray()) {
        for(valence.JSONParse item : node.asList()) {
            explore(item, depth + 1);
        }
    }
}

valence.JSONParse root = new valence.JSONParse('{"menu":{"id":"file","value":"File",
↪"popup":{"menuitem":[{"value":"New","onclick":"CreateNewDoc()"},{"value":"Open",
↪"onclick":"OpenDoc()"},{"value":"Close","onclick":"CloseDoc()"}]}}}');

explore(root, 0);
```

Result

```
13:37:23.34 (39921697)|USER_DEBUG|[3]|DEBUG|**file
13:37:23.34 (40158891)|USER_DEBUG|[3]|DEBUG|**File
13:37:23.34 (40964502)|USER_DEBUG|[3]|DEBUG|*****New
13:37:23.34 (41091905)|USER_DEBUG|[3]|DEBUG|*****CreateNewDoc()
13:37:23.34 (41310876)|USER_DEBUG|[3]|DEBUG|*****Open
13:37:23.34 (41477025)|USER_DEBUG|[3]|DEBUG|*****OpenDoc()
13:37:23.34 (41820815)|USER_DEBUG|[3]|DEBUG|*****Close
13:37:23.34 (41952080)|USER_DEBUG|[3]|DEBUG|*****CloseDoc()
```

5.34.7 Primitive Value Extraction

A parser is no good if you can't get to the juicy stuff it's wrapped round!

There are a number of methods to pull primitive values out of JSONParse nodes. In general we followed the conventions in the native JSONParser class, except in a few places where we supported a broader / more flexible set of behaviors. For example, you can build Date or Time instances from Long values.

```
public Blob getBlobValue() {}

public Boolean getBooleanValue() {}

public Datetime getDatetimeValue() {}

public Date getDateValue() {}

public Decimal getDecimalValue() {}

public Double getDoubleValue() {}

public Id getIdValue() {}

public Integer getIntegerValue() {}

public Long getLongValue() {}

public String getStringValue() {}

public Time getTimeValue() {}

public Object getValue() {}
```

Disclaimers

This helper class only reads JSON, it does not write it. For writing, may we suggest using the native and perfectly serviceable JSON.serialize()!

5.35 LinkContext

This is a class that is full of information that might be useful to your Adapter or Filter while it is executing. It is an example of the Context Object pattern.

LinkContext has information about the Link that is currently running. This is a key object that allows custom extensions to be as dynamic as possible, reacting to run-time information for the currently-executing Link and operating under that context.

For example, you can use LinkContext to know which table you should be querying from an external system, or which fields to get from that table. A Filter can use LinkContext to see what mappings are active and if any of them have had configurations defined for this Filter.

Because LinkContext is tightly coupled and intrinsic to a running Link, you'll find it passed as a parameter to almost every interface method so you always have it handy.

5.35.1 Properties

Data Type	Class Property	Description
String	linkName	The API name of the Link that is running. Comes from the DeveloperName cMDT record for the Link.
String	linkSourceName	The API table name of the source Table, if one has been defined.
String	linkSourceLabel	The user-friendly label of the source Table, if one has been defined.
String	linkSourceTableDetails	Additional details about the source table that the source Adapter wanted to store.
String	linkTargetName	The API table name of the target Table, if one has been defined.
String	linkTargetLabel	The user-friendly label of the target Table, if one has been defined.
String	linkTargetTableDetails	Additional details about the target table that the target Adapter wanted to store.
Boolean	testingMode	True if this Link is running in Test Mode. No TargetAdapter should write to its backing system in Test Mode.
Boolean	isReplay	True if this Link run is a replay of serialized failed records.
String	sourceAdapterNamespace	The namespace of the Adapter that is being used as the source Adapter.
String	sourceAdapterClassName	The class name of the Adapter that is being used as the source Adapter.
String	targetAdapterNamespace	The namespace of the Adapter that is being used as the target Adapter.
String	targetAdapterClassName	The class name of the Adapter that is being used as the target Adapter.
String	recordSnapshotLoggingLevel	The user's selected level of logging. Picklist with possible values: All,Errors/Warnings,Errors Only,None
Integer	batchSizeLimit	The maximum number of records that can be processed in each batch for this Link.
DateTime	lastSuccessfulSync	The timestamp of the last time this Link successfully synced, or null if it has never successfully synced before.
String	lastSuccessfulCursor	A value you stored, like a bookmark, during the last time this Link successfully synced, or null if it has never successfully synced before.
Datetime	now	The timestamp that will be saved to the Sync Event as the retrieval timestamp. We recommend SourceAdapters fetch records with modification timestamps after **lastSuccessfulSync** and before **now**.
Map<String, *Mapping*>	mappings	All active mappings for this Link. *Inactive mappings will not be in this collection.*
List<*FieldPath*>	suggestedQueryFields	A filtered list of what Valence thinks you should query for if you are a source Adapter working with this context.

5.35.2 Test Coverage

Because LinkContext is so intrinsic to all facets of working with Valence, you will use an instance of one in just about every test method you write. Luckily, they're trivially simple to set up and work with. Create one and set any properties you'll need in your test.

```
    // set up a new, empty instance
valence.LinkContext context = new valence.LinkContext();

// set properties that you need, for example testing a source Adapter
    context.linkSourceName = 'SomeSourceTable';
    context.batchSizeLimit = 200;

    Test.startTest();
    MyAdapter adapter = new MyAdapter();
    valence.FetchStrategy strategy = adapter.planFetch(context);
    Test.stopTest();
```

5.36 Mapping

Mapping is a special Apex class that gives *Adapters* and *Filters* info about the mappings a user has defined for the *Link*.

Mappings can be found as a property inside a *LinkContext*.

The Mapping class is pretty straightforward, the only thing that is a little special is the **configuration** property. It is contextually-sensitive, so for example if you are inspecting the Mappings from inside your custom *Filter*, if there's a value inside **configuration** it is because a user has specifically set a configuration for YOUR Filter on THIS Link, so go ahead and use it.

For more information about mapping configurations, check out *ConfigurablePerMappingFilter*.

5.36.1 Properties

Data Type	Class Property	Description
String	name	A unique name for this mapping, comes from the DeveloperName field of the corresponding cMDT record.
Field Path	sourceFieldPath	Reference to the specific Field that is being queried from; different from a property path.
List<String>	sourcePropertyPath	A representation of which data values to read from the original record properties.
Field Path	targetFieldPath	Reference to the specific Field that is being written to; different from a property path.
List<String>	targetPropertyPath	A representation of where to write to within the active record properties.
String	configuration	Configuration information, if any has been specified for this mapping in this context.

Field Path vs Property Path

Field Path and *Property Path* are similar concepts and the distinction between the two is best explained with some examples.

When we are thinking about a schema shape, it's an abstract tree of which fields might be present on a hypothetical record. A FieldPath represents a Field's position in this tree.

However, when we have an actual, concrete record that we are working with, we want to be more specific when it comes to nested lists. That's where PropertyPath comes in.

Here's an example Mapping:

```
{'fieldPath': ['contacts','firstname'], 'propertyPath' : ['"contacts"', '*', '"firstname"
↪']}
```

Here's another Mapping with the same field path, but a different property path:

> {'fieldPath': ['contacts','firstname'], 'propertyPath' : ['"contacts"', '0,1', '"firstname"']}

Tip: Typically you will be working with property paths in most use cases; field paths are more specialized for certain scenarios where it is useful to think in terms of the fields separate to some degree from the records (for example *when a source adapter is deciding which fields to query for*).

5.36.2 Test Coverage

There are methods for creating Mapping instances for test purposes in **ValenceTestUtil**:

```
global static Mapping createTestMapping(String name, String conciseSourcePropertyPath,␣
↪String conciseTargetPropertyPath, String configuration);

global static Mapping createTestMapping(String name, List<String>␣
↪normalizedSourcePropertyPath, List<String> normalizedTargetPropertyPath, String␣
↪configuration);
```

You can set up Mapping instances to use in your test classes as follows.

```
// inside your test class

Map<String, valence.Mapping> mappings = new Map<String, valence.Mapping>();
mappings.put('first', ValenceTestUtil.createTestMapping('first', 'ParentAccount.
↪AccountName', 'MyParentAccountName', null));

// set up a new, empty instance of LinkContext
valence.LinkContext context = new valence.LinkContext();

// set properties that you need, for example testing a source Adapter
context.linkSourceName = 'SomeSourceTable';
context.batchSizeLimit = 200;
context.mappings = mappings;

// etc
```

5.37 Property Node

PropertyNode is a facade that we wrap around the underlying data structures of *RecordInFlight*, allowing for easier, more intuitive interactions when both reading from and writing to the record.

Typically you won't work with PropertyNode directly, but instead by invoking methods on RecordInFlight that accept *Property Path* parameters.

If your extension really needs to get into the guts of a property tree and either manipulate values in place or do some kind of heavy lifting, you can get an instance of PropertyNode to work with like so:

```
// RecordInFlight instance methods
global PropertyNode getOriginalPropertiesRoot();
global PropertyNode getPropertiesRoot();
```

Much like *JSONParse*, PropertyNode treats the underlying data as a tree and wraps each node in its own instance of PropertyNode (thus the class name). You can use property path notation to traverse the tree, retrieving specific nodes of interest. From a given node you can traverse further into that subtree.

5.37.1 A Common Use Case

One of the most likely reasons you'll want to work with PropertyNode directly is when you want to manipulate a value at an arbitrary location in a record's property tree, and you don't actually care if value X is three layers down, or two, or has key A or key B.

Maybe the job of your extension in this moment is to capitalize stuff. So you have zero opinion about where to find stuff to capitalize, just that you capitalize whatever you're told to. The user configures the "when" and "where", and you handle the "what".

```
// handed a property path value from Valence because of a configuration or mapping etc
String configuredPropertyPath = 'offices[*].employees[2,5].title';

PropertyNode root = record.getPropertiesRoot();

for(PropertyNode node : root.collectAllNodes(configuredPropertyPath)) {
        node.setValue(((String)node.getValue()).capitalize());
}

// we capitalized the title on the third and sixth employees, and we did that on every␣
↪office
```

This is really powerful because you were able to do your job and remain agnostic about the actual data shape of the record.

5.37.2 Definition

```
global Object getValue();

global void setValue(Object newValue);

global List<String> getPath(); // concrete property path to this exact node

```

(continues on next page)

(continued from previous page)

```
global Boolean hasProperty(String concisePropertyPath); // true if a property exists at␣
↪this path

global Boolean hasProperty(List<String> normalizedPropertyPath); // true if a property␣
↪exists at this path

global Boolean hasProperty(String concisePropertyPath, Boolean notNull); // true if a␣
↪property exists and its value is not null

global Boolean hasProperty(List<String> normalizedPropertyPath, Boolean notNull); //␣
↪true if a property exists and its value is not null

global Object getPropertyValue(String concisePropertyPath);

global Object getPropertyValue(List<String> normalizedPropertyPath);

global List<Object> getPropertyValues(String concisePropertyPath);

global List<Object> getPropertyValues(List<String> normalizedPropertyPath);

global void setPropertyValue(String concisePropertyPath, Object value);

global void setPropertyValue(List<String> normalizedPropertyPath, Object value);

global void setPropertyValues(String concisePropertyPath, List<Object> values);

global void setPropertyValues(List<String> normalizedPropertyPath, List<Object> values);

global List<PropertyNode> collectAllNodes(String concisePropertyPath);

/**
 * Recursively gather all nodes in the tree that the passed property path points to.␣
↪Resolves indefinite
 * paths into potentially multiple nodes, each of which will be parsed and collected.
 *
 * When this method is finished you will receive a final list of all nodes that matched␣
↪the original path,
 * and you can go ahead and extract the values to get your real result.
 *
 * @param propertyPath A property path using our property path syntax
 *
 * @return All nodes at any level at or beneath this one in the tree that match the␣
↪property path
 */
global List<PropertyNode> collectAllNodes(List<String> normalizedPropertyPath);
```

5.38 RecordInFlight

RecordInFlight represents a single record as it moves through the Valence framework. RecordInFlight holds not just the record properties but also metadata such as errors and warnings associated with the record.

5.38.1 Record Data

We think of each data record as a tree with key-value pairs, with nested subtrees if needed. RecordInFlight internally uses two of these tree structures, one with the original properties that it was first created with, one with the properties as they are manipulated and changed by *Filters*.

Typically you will interact with the properties tree, unless you specifically are interested in reading a value from what the record looked like originally.

RecordInFlight Data Methods

```
// constructor
global RecordInFlight(Map<String, Object> originalProperties);

// constructor
global RecordInFlight(Map<String, Object> originalProperties, String operation);


global Boolean hasOriginalProperty(String concisePropertyPath);

global Boolean hasOriginalProperty(List<String> normalizedPropertyPath);

global Boolean hasOriginalProperty(String concisePropertyPath, Boolean notNull); // is␣
↪there a property, and is its value also not null?

global Boolean hasOriginalProperty(List<String> normalizedPropertyPath, Boolean notNull);
↪ // is there a property, and is its value also not null?


global Boolean hasProperty(String concisePropertyPath);

global Boolean hasProperty(List<String> normalizedPropertyPath);

global Boolean hasProperty(String concisePropertyPath, Boolean notNull); // is there a␣
↪property, and is its value also not null?

global Boolean hasProperty(List<String> normalizedPropertyPath, Boolean notNull); // is␣
↪there a property, and is its value also not null?


global Object getOriginalPropertyValue(String concisePropertyPath); // convenience␣
↪method for when you know for sure there's only one value to get

global Object getOriginalPropertyValue(List<String> normalizedPropertyPath); //␣
↪convenience method for when you know for sure there's only one value to get
```

(continues on next page)

(continued from previous page)

```
global List<Object> getOriginalPropertyValues(String concisePropertyPath);

global List<Object> getOriginalPropertyValues(List<String> normalizedPropertyPath);


global Object getPropertyValue(String concisePropertyPath); // convenience method for␣
↪when you know for sure there's only one value to get

global Object getPropertyValue(List<String> normalizedPropertyPath); // convenience␣
↪method for when you know for sure there's only one value to get

global List<Object> getPropertyValues(String concisePropertyPath);

global List<Object> getPropertyValues(List<String> normalizedPropertyPath);


global void setOriginalPropertyValue(String concisePropertyPath, Object value);

global void setOriginalPropertyValue(List<String> normalizedPropertyPath, Object value);

global void setOriginalPropertyValues(String concisePropertyPath, List<Object> values);

global void setOriginalPropertyValues(List<String> normalizedPropertyPath, List<Object>␣
↪values);


global void setPropertyValue(String concisePropertyPath, Object value);

global void setPropertyValue(List<String> normalizedPropertyPath, Object value);

global void setPropertyValues(String concisePropertyPath, List<Object> values);

global void setPropertyValues(List<String> normalizedPropertyPath, List<Object> values);
```

Working with Record Data

```
Map<String,Object> originalProps = new Map<String,Object>{
        'firstName' => 'Tom',
        'lastName' => 'Sinatra',
        'company' => new Map<String, Object> {
                'name' => 'Acme',
                'location' => 'USA'
        }
};

// creating a RecordInFlight
valence.RecordInFlight tom = new valence.RecordInFlight(originalProps);

// accessing the original properties
System.assertEquals('Sinatra', tom.getOriginalPropertyValue('lastName'));
```

(continues on next page)

(continued from previous page)

```
System.assertEquals('Acme', tom.getOriginalPropertyValue('company.name'));

// accessing properties that are being modified during the Link run
tom.getPropertyValue('discountCode');
tom.setPropertyValue('seatPreference', 'Aisle');
```

5.38.2 Reporting Errors and Warnings

Any *Adapter* or *Filter* that touches a RecordInFlight can mark that record as having an issue of some kind. Adding errors and warnings to a record has different outcomes depending on how the Valence user has configured the Link.

```
global Boolean hasWarnings();

global void addWarning(String warning);

global void addWarning(String warning, Exception e);

global Boolean hasErrors();

global void addError(String error);

global void addError(String error, Exception e);
```

5.38.3 Ignoring Records

If you would like to skip the processing of certain records you can call the ignore method. This removes the record from further processing, and will also track ignore reasons and counts and surface those in the interface for an admin to see.

```
global Boolean isIgnored();

global void ignore(String reason);
```

5.38.4 Operation

Every RecordInFlight has an **operation**, which is just a string value that suggests an action to the *TargetAdapter*. The default operation value is "upsert". We recommend every SourceAdapter and TargetAdapter support at a minimum these two operations:

- upsert
- delete

You are welcome to create custom operation values as long as the TargetAdapter you are working with knows how to handle them.

```
global void setOperation(String operation);

global String getOperation();
```

5.38.5 Metadata

There are a few additional metadata properties that can be set or read to better understand what is happening with this particular RecordInFlight.

```
// Our first-party Adapters that work with Salesforce orgs will always populate a␣
↪Salesforce Id value that you can use if you need it
global Id getSalesforceId();

// TargetAdapters should set this value
global void setSuccess(Boolean newValue);

// was this RecordInFlight successful in being delivered to its destination
global Boolean isSuccess();

// TargetAdapters should set this value
global void setCreated(Boolean newValue);

// did this RecordInFlight cause a brand new record to be created (true) or did it␣
↪update or delete an existing record (false)
global Boolean isCreated();
```

5.39 Table

The Table class represents a possible source or target for a Link. It is analogous to a database table or Salesforce object.

Hint: Keep in mind that a Table doesn't have to correspond to an actual database table. You can create Table instances as logical constructs, perhaps to represent different parts of the same API.

A Table can be specified in a *Link* as either the **source** or **target** of records. However, this is not required for a Link to function, and in fact it's perfectly fine for an *Adapter* to be "schema-less". Creating Table instances goes hand-in-hand with the *SchemaAdapter* interface.

Tables are constructed using a builder pattern that uses a fluent interface to make it simple to construct Table instances with varying degrees of complexity.

You get a TableBuilder by calling Table.create() and passing the API name of the Table you are constructing. This is the only required property, but it's an important one. This string value is used when asking your Adapter for the Fields that be found for a given table. You will also likely use this table name value when your Adapter is figuring out which table the user wants to get records from, or send records to.

5.39.1 Simple Table Example

```
valence.Table companiesTable = valence.Table.create('company').withLabel('Companies').
↪build();
```

5.39.2 Properties

Data Type	Class Property	Builder Method	Description
String	name		Required. Used to retrieve lists of Fields.
String	label	withLabel	User-friendly label for this table.
String	description	withDescription	Description of the table's use.
String	tableDetails	withTableDetails	Additional details you might need later. See *Additional Table Details*.
Boolean	isCreateable	setCreateable	True if new records can be created in this table (default: false).
Boolean	isReadable	setReadable	True if records can be read from this table (default: true).
Boolean	isUpdateable	setUpdateable	True if existing records can be updated in this table (default: false).
Boolean	isDeletable	setDeletable	True if records can be deleted from this table (default: false).
Boolean	isEditable	setEditable	True if either isCreateable or isUpdateable have been set. Can also be set independently (default: false).

5.40 ChainFetchAdapter

This interface allows your *Adapter* to indefinitely chain fetchRecord() calls, each in its own execution context.

This is an extension interface to *SourceAdapterForPull*. It complements the **IMMEDIATE** *FetchStrategy* by allowing you to repeatedly call your fetchRecords() method, each time in a new execution context with a new scope.

This interface will be tremendously useful to you if you are fetching records from an external system that cannot predict in advance how many records it will be giving you. Some APIs simply say "I have more records to give you, please query again with this token/page/url", and you have to keep calling them until you exhaust the records. This interface helps in that circumstance.

If you implement this interface, right after fetchRecords() is called—and in the same execution context—getNextScope() will be called. Store whatever information you need in the scope and it will be passed to the next invocation of fetchRecords().

Note: Chaining will continue until you return a null from getNextScope().

Warning: The very first time fetchRecords() is called, a null scope will be passed to your adapter. This is because getNextScope() has never yet been called. You are expected to initialize appropriate scope state in your planFetch().

So, to be clear, the order of execution will be:

1. planFetch()
2. fetchRecords(null scope)
3. getNextScope()
4. –NEW EXECUTION CONTEXT–
5. fetchRecords(scope returned by #3)

6. getNextScope()
7. –NEW EXECUTION CONTEXT–
8. fetchRecords(scope returned by #6)
9. getNextScope()
10. …and so on, indefinitely, until you return a null value from getNextScope()

5.40.1 Definition

```
/**
 * Implement this interface if you are fetching data from an external API that can't tell␣
↪you in advance how
 * many records will be returned. Implementing this Adapter allows Valence to alternate␣
↪asking your Adapter
 * for records with requests for the next scope to use.
 */
global interface ChainFetchAdapter extends SourceAdapterForPull {

        /**
         * @return A scope object that will be passed back to you on the next call to␣
↪FetchRecords.
         */
        Object getNextScope();
}
```

5.41 ConfigurableSourceAdapter

This interface allows your *Adapter* to be configurable by Valence users when it is used as a source Adapter. What "configurable" means is entirely up to you. Maybe you need an additional piece of security information, or perhaps you're going to let users add restrictions on which records your Adapter fetches.

Allowing users to alter the behavior of your adapter across different Links gives them a lot of flexibility.

Depending on which interfaces you have implemented, your Adapter may be a source adapter, a target adapter, or both. There is a different interface if you want to *make your adapter configurable as a target adapter*.

The explainConfiguration() method gives you an opportunity to share with users a contextually-aware description of what your configuration is doing. Don't just describe in abstract what can be configured, but rather use the current configuration values the user has chosen to explain to them what effect this will have on your Adapter.

Note: For a deeper look at configurable extensions in Valence, read our guide on *Configurability*.

Note: For example usage, see how we allow users to change their upsert field with our example in *ConfigurableTargetAdapter*.

5.41.1 Definition

```
/**
 * Implement this interface if you would like your SourceAdapter to be configurable by␣
↪Users to behave differently for each Link that uses it.
 */
global interface ConfigurableSourceAdapter {

        /**
         * You can use your own Lightning component to let Users build and edit your␣
↪configuration. If you want to do this, return the fully qualified
         * name of your component, which looks like this:
         *
         * valence:MyAwesomeAdapterConfigurator
         *
         * Make sure your component is set to global so that Valence can instantiate it.
         *
         * @param context Information about this Link
         *
         * @return The name of your Lightning component that will handle configuration,␣
↪or null if you don't need your own component
         */
        String getSourceConfigurationLightningComponent(LinkContext context);

        /**
         * If you don't need or don't want to use your own Lightning Component, you can␣
↪simply describe your configuration shape and we will present
         * the user with some basic input fields to populate values in your␣
↪configuration.
         *
         * @param context Information about this Link
         *
         * @return A serialized JSON object describing your configuration data structure,
↪ or null if you use your own component
         */
        String getSourceConfigurationStructure(LinkContext context);

        /**
         * Given configuration data, return a user-friendly paragraph that explains how␣
↪this specific configuration
         * is going to be used by your class and what effect that will have on the Link␣
↪being run.
         *
         * We show this in the user interface to help Users understand the impact of␣
↪their configurations.
         *
         * @param context Information about this Link
         * @param configurationData Configuration data in JSON format, or in whatever␣
↪format your custom configuration component gave us
         *
         * @return A human-readable and friendly explanation that specifically reflects␣
↪and explains the configuration passed.
         */
```

(continues on next page)

(continued from previous page)

```
        String explainSourceConfiguration(LinkContext context, String configurationData);

        /**
         * Sets configuration data for your Adapter. This is the first method called on␣
↪your Adapter during Link execution.
         *
         * @param context Information about this Link and the current execution of it.
         * @param configurationData Configuration data in JSON format, or in whatever␣
↪format your custom configuration component gave us
         */
        void setSourceConfiguration(LinkContext context, String configurationData);
}
```

5.42 ConfigurableTargetAdapter

This interface allows your *Adapter* to be configurable by Valence users when it is used as a target Adapter. What that means is entirely up to you.

Allowing users to alter the behavior of your adapter across different Links gives them a lot of flexibility. As an example, we use this on our Valence local Salesforce adapter to allow users to pick the upsert field they would like to use for each Link.

Depending on which interfaces you have implemented, your Adapter may be a source adapter, a target adapter, or both. There is a different interface if you want to *make your adapter configurable as a source adapter*.

The explainConfiguration() method gives you an opportunity to share with users a contextually-aware description of what your configuration is doing. Don't just describe in abstract what can be configured, but rather use the current configuration values the user has chosen to explain to them what effect this will have on your Adapter.

Note: For a deeper look at configurable extensions in Valence, read our guide on *Configurability*.

5.42.1 Definition

```
/**
 * Implement this interface if you would like your TargetAdapter to be configurable by␣
↪Users to behave differently for each Link that uses it.
 */
global interface ConfigurableTargetAdapter {

        /**
         * You can use your own Lightning component to let Users build and edit your␣
↪configuration. If you want to do this, return the fully qualified
         * name of your component, which looks like this:
         *
         * valence:MyAwesomeAdapterConfigurator
         *
         * Make sure your component is set to global so that Valence can instantiate it.
         *
         * @param context Information about this Link
```

(continues on next page)

(continued from previous page)

```
         *
         * @return The name of your Lightning component that will handle configuration,␣
↪or null if you don't need your own component
         */
        String getTargetConfigurationLightningComponent(LinkContext context);

        /**
         * If you don't need or don't want to use your own Lightning Component, you can␣
↪simply describe your configuration shape and we will present
         * the user with some basic input fields to populate values in your␣
↪configuration.
         *
         * @param context Information about this Link
         *
         * @return A serialized JSON object describing your configuration data structure,
↪ or null if you use your own component
         */
        String getTargetConfigurationStructure(LinkContext context);

        /**
         * Given configuration data, return a user-friendly paragraph that explains how␣
↪this specific configuration
         * is going to be used by your class and what effect that will have on the Link␣
↪being run.
         *
         * We show this in the user interface to help Users understand the impact of␣
↪their configurations.
         *
         * @param context Information about this Link
         * @param configurationData Configuration data in JSON format, or in whatever␣
↪format your custom configuration component gave us
         *
         * @return A human-readable and friendly explanation that specifically reflects␣
↪and explains the configuration passed.
         */
        String explainTargetConfiguration(LinkContext context, String configurationData);

        /**
         * Sets configuration data for your Adapter. This is the first method called on␣
↪your Adapter during Link execution.
         *
         * @param context Information about this Link and the current execution of it.
         * @param configurationData Configuration data in JSON format, or in whatever␣
↪format your custom configuration component gave us
         */
        void setTargetConfiguration(LinkContext context, String configurationData);
}
```

5.42.2 Example Usage

```
public MyAdapter implements valence.ConfigurableTargetAdapter {

    private Schema.SObjectField upsertField = null;

    public String getTargetConfigurationLightningComponent(valence.LinkContext context) {
        return 'valence:MyCustomConfigurator';
    }

    public String getTargetConfigurationStructure(valence.LinkContext context) {
        return null;
    }

    public String explainTargetConfiguration(valence.LinkContext context, String␣
↪configurationData) {

        String defaultMessage = 'Not configured; will default to letting Salesforce use␣
↪Id as the upsert field.';

        if(String.isBlank(configurationData))
            return defaultMessage;

        try {
            Map<String, Object> config = (Map<String, Object>)JSON.
↪deserializeUntyped(configurationData);

            // set the upsertField by parsing some config data and using that with␣
↪Schema describe to find a field token
            if(config.containsKey('upsertField')) {
                String upsertFieldName = String.valueOf(config.get('upsertField'));
                upsertField = describeSObject(context.linkTargetName).fields.getMap().
↪get(upsertFieldName);
                if(upsertField != null)
                    return 'This Adapter will use the <strong>' + upsertField.
↪getDescribe().label + '</strong> field as the unique external Id when upserting␣
↪records into Salesforce.';
                else
                    return '<p class="slds-theme_warning">This Adapter is configured to␣
↪upsert using <' + upsertFieldName + '> but that field does not exist in <strong>' +␣
↪context.linkTargetLabel + '</strong>.</p>';
            }
            else
                return defaultMessage;
        }
        catch(Exception e) {
            return '<p class="slds-theme_error">The configuration for this Adapter is␣
↪malformed.</p>';
        }
    }

    public void setTargetConfiguration(valence.LinkContext context, String␣
↪configurationData) {
```

(continues on next page)

(continued from previous page)

```
        if(String.isBlank(configurationData))
            return;

        try {
            Map<String, Object> config = (Map<String, Object>)JSON.
↪deserializeUntyped(configurationData);

            // set the upsertField by parsing some config data and using that with␣
↪Schema describe to find a field token
            if(config.containsKey('upsertField'))
                upsertField = describeSObject(context.linkTargetName).fields.getMap().
↪get(String.valueOf(config.get('upsertField')));
        }
        catch(Exception e) {
            throw new AdapterException('Failed to parse Adapter configuration.', e);
        }
    }
}
```

5.43 DelayedPlanningAdapter

This interface helps your *Adapter* to delay planning a fetch.

This is helpful when the data source you are working with is asynchronous and you have to set up a fetch then check back later to get the data.

This is an extension interface to *SourceAdapterForPull*. It complements the **DELAY** *FetchStrategy* by allowing you to repeatedly call your planFetchAgain() method, each time in a new execution context after some delay, with whatever scope you decide to carry between executions.

If your Adapter returns the **DELAY** FetchStrategy, it must implement this interface.

Note: Chaining will continue until you return a FetchStrategy other than **DELAY** from your planFetchAgain() method. Returning **DELAY** is telling Valence you're still not ready and you need some more time.

Warning: The very first time planFetch() is called, you can return any *FetchStrategy* you like. If you return **DELAY**, this interface comes into effect. Valence will wait for a certain amount of time, then call planFetchAgain(). Note that the normal planFetch() method is never called more than once. After the first planFetch(), all additional attempt to plan will be routed to planFetchAgain(). This is so that you can set up a scope and pass it to yourself over and over, for example the jobId for whatever async process you started in your original planFetch() call.

So, to be clear, an example order of execution might be:

1. planFetch() [returning **DELAY**]
2. –NEW EXECUTION CONTEXT–
3. planFetchAgain() [returning **DELAY** again]
4. –NEW EXECUTION CONTEXT–
5. planFetchAgain() [returning **DELAY** again]

6. –NEW EXECUTION CONTEXT–
7. planFetchAgain() [returning **IMMEDIATE**]
8. fetchRecords() [null scope, per usual **IMMEDIATE** behavior]

Tip: You can return any FetchStrategy from planFetchAgain() that you would return from planFetch().

5.43.1 Definition

```
/**
 * Implement this interface if your SourceAdapter can't resolve its planning in a single␣
↪call to planFetch() and needs more time.
 *
 * Companion interface to the DELAY FetchStrategy.
 */
global interface DelayedPlanningAdapter extends SourceAdapterForPull {

        /**
         * Called after planFetch() has been called once and returned a DELAY␣
↪FetchStrategy. This method is then called after
         * some amount of time. You can chain this method again and again if you return␣
↪a DELAY strategy from this method.
         *
         * Use this if planning your fetch is asynchronous or takes some time so that␣
↪you can wait to start fetching until you're actually ready.
         *
         * @param context Information about this Link and the current execution of it.
         * @param scope Any additional details the original call to planFetch() gave us␣
↪for safekeeping
         *
         * @return The FetchStrategy to use
         */
        FetchStrategy planFetchAgain(LinkContext context, Object scope);
}
```

Tip: Passing a *null* as your minutes parameter for valence.FetchStrategy.delay(minutes, scope) will let Valence decide how long to wait before calling planFetchAgain(), which will usually be somewhere between 5 seconds and 75 seconds.

5.44 LazyLoadSchemaAdapter

A functionality extension of *SchemaAdapter*, this interface provides structure for loading partial schemas within a single *Table*.

When building *Field* instances that are Lists or Maps, a flag can be set to mark that Field as "lazy" and available for lazy loading.

Loading part of the schema later in time improves performance and reduces overhead.

If you mark a Field as lazy, Valence may (or may not) call your Adapter to get details about that Field. If that happens, you will be passed info about the Field of interest and should return a List<Field> **where each item in that array is an immediate child** of the Field that was inquired about.

The method in this interface uses an instance of *Field Path* to describe what Field should be lazily-loaded.

If you are dynamically inspecting external system schema, you will likely also want to implement *NamedCredential-Adapter*.

To learn more about how lazy loading is used, check out our *Schema* documentation.

5.44.1 Definition

```
/**
 * Implementers of this interface can return lazy Fields from the normal SchemaAdapter␣
↪call. This interface
 * allows Valence to retrieve the children of those Fields later in time.
 */
global interface LazyLoadSchemaAdapter extends SchemaAdapter {

	List<Field> getLazyFieldChildren(String tableApiName, FieldPath path);
}
```

5.45 NamedCredentialAdapter

This interface allows your *Adapter* to be given a NamedCredential to use when making callouts.

For more information about using NamedCredentials for callouts, check out the Salesforce documentation. This interface has one method, where you are passed the string api name for the NamedCredential that has been selected for use by the Valence user. You can use this name directly in your callout URL.

The setNamedCredential() method is called, **when needed**, before any other methods your Adapter is implementing from other interfaces, such as planFetch() or fetchRecords().

The **when needed** means that the method will be called before say, a *getTables()* call if your Adapter registration record is marked as **RequiresNamedCredentialForSchema__c**. However, if **RequiresNamedCredential-ForSchema__c** is unchecked, setNamedCredential() will not be called before getTables(). Basically, setNamedCredential() is contextually-sensitive and will only be called when it is needed.

5.45.1 Definition

```
/**
 * Implement this interface if your Adapter needs a NamedCredential in order to␣
↪communicate with its data source. See
 * the Adapter custom metadata type for additional configuration options around when the␣
↪NamedCredential comes into effect.
 */
global interface NamedCredentialAdapter extends Adapter {

        /**
         * Gives you the NamedCredential name that you will need in order to do an Apex␣
↪callout or get information about
         * the endpoint the User would like to talk to using your adapter.
         *
         * @param namedCredentialName The API name of a NamedCredential defined in this␣
↪Salesforce org
         *
         * @see https://developer.salesforce.com/docs/atlas.en-us.apexcode.meta/apexcode/
↪apex_callouts_named_credentials.htm
         */
        void setNamedCredential(String namedCredentialName);
}
```

5.45.2 Example Usage

```
private String namedCredentialName = null;

public void setNamedCredential(String namedCredentialName) {
    this.namedCredentialName = namedCredentialName;
}

public List<valence.RecordInFlight> fetchRecords(valence.LinkContext context, Object␣
↪scope) {

    HttpRequest req = new HttpRequest();
    req.setMethod('POST');
    req.setEndpoint('callout:' + namedCredentialName);
    req.setHeader('Content-Type', 'text/xml');
    HttpResponse resp = new Http().send(req);

    // process response and return RecordInFlight instances
}
```

5.46 SchemaAdapter

This interface allows Valence to interrogate your custom *Adapter* for information about what tables and fields are accessible in an external system.

You will sometimes hardcode your schema into your Adapter, for example if you're working against an API that rarely changes shape (or changing shape would require code changes anyways). Generally, we recommend that you try to make your schema inspection dynamic such that if the database structure changes your Adapter can report on the new structure without having its coding updated. Think about how Salesforce custom fields and objects are so easily added, and information about them exposed through the Metadata API or Schema Describe. Try to do that with your Adapter.

For additional documentation, see the pages on *Table* and *Field*.

If you are dynamically inspecting external system schema, you will likely also want to implement *NamedCredential-Adapter*.

To learn more about how Valence uses schema information, check out our *Schema* page.

Hint: Two valuable things to be aware of:

1. **Not every** Adapter has to implement SchemaAdapter. It is perfectly fine for an Adapter to not have a schema. The Adapter will be treated by Valence as if it only accesses a single, unnamed table. Users will not be shown a choice of tables. Fields for mapping will be discovered by Valence (see #2).
2. Whether you implement SchemaAdapter or not, Valence always inspects record shape as records go by and will surface potential fields to Valence users as mapping targets. If you miss fields in your getFields() call, Valence can still spot them.

5.46.1 Definition

```
/**
 * Implement this interface if your Adapter has a structured schema and you want to␣
↪allow the User to select
 * from a list of specific tables and fields that they can interact with.
 */
global interface SchemaAdapter extends Adapter {

        /**
         * We will interrogate your adapter and ask it what tables can be interacted␣
↪with.
         *
         * @return A List of Table definitions that will be provided to Users.
         */
        List<Table> getTables();

        /**
         * A natural follow-on from getTables, we will interrogate your adapter to
         * find out which fields can be interacted with on a table.
         *
         * @param tableApiName The specific table a User is interested in, comes from␣
↪your list returned by getTables()
         *
         * @return A List of Field definitions that will be provided to Users for␣
```

(continues on next page)

(continued from previous page)

```
↪consideration.
         */
        List<Field> getFields(String tableApiName);
}
```

5.46.2 Example Usage - Hardcoded

```
public List<valence.Table> getTables() {
    return new List<valence.Table>{
        valence.Table.create('Company').withLabel('Company').withDescription(
↪'Definitions of businesses.').build(),
        valence.Table.create('Person').withLabel('Person').withDescription('People that␣
↪are associated with a business.').build()
    };
}
```

5.46.3 Example Usage - Dynamic

```
public List<valence.Table> getTables() {

    // Send the request
    Map<String,String> serverTables = fetchTablesFromExternalServer();

    List<valence.Table> tables = new List<valence.Table>();

    // iterate through response elements
    for(String key : serverTables.keySet()) {
        tables.add(
            valence.Table.create(key)
                .withLabel(serverTables.get(key))
                .build()
        );
    }

    return tables;
}
```

5.47 SourceAdapterForPull

This interface allows an *Adapter* to be the source in a Pull Link, fetching records in batches from a source data store.

This is a very commonly-used interface; you will likely also want to consider implementing *NamedCredentialAdapter* and *SchemaAdapter*.

5.47.1 Definition

```
/**
 * Implement this interface if your Adapter knows how to fetch its own records. The most␣
↪common
 * example of this is an Adapter that calls an external system API to retrieve records.
 */
global interface SourceAdapterForPull extends SourceAdapter {

        /**
         * This method helps you to scale seamlessly to fetch large numbers of records.␣
↪We do this by splitting requests
         * out into separate execution contexts, if need be.
         *
         * Valence will call planFetch() on your Adapter first, and then start calling␣
↪fetchRecords(). The number of times
         * fetchRecords() is called depends on what you return from planFetch(). Every␣
↪call to fetchRecords() will be in
         * its own execution context with a new instance of your Adapter, so you'll lose␣
↪any state you have in your class.
         *
         * @param context Information about this Link and the current execution of it.
         *
         * @return An instance FetchStrategy that will tell the Valence engine what␣
↪should happen next
         */
        FetchStrategy planFetch(LinkContext context);

        /**
         * Second, we will call this method sequentially with scopes you gave us in␣
↪response to planPush(). We give you your
         * scope back so you can use it as needed.
         *
         * If you need to mark errors as failed or warning, use the addError() and␣
↪addWarning() methods on RecordInFlight.
         *
         * @param context Information about this Link and the current execution of it.
         * @param scope A single scope instance from the List of scopes returned by␣
↪planFetch()
         *
         * @return All of the records that have been updated since the timestamp passed␣
↪inside LinkContext.
         */
        List<RecordInFlight> fetchRecords(LinkContext context, Object scope);
}
```

Tip: Any Adapter that implements this interface should also immediately implement *SourceAdapterScopeSerializer*. It's easy to implement and it allows Links that use your source Adapter to run in parallel mode, which is much, much faster.

5.47.2 Behavior

Records are retrieved in a two-step process:

1. Planning

First, there is a planning step. This allows Valence and your Adapter to negotiate the exchange of records that is about to happen. Valence gives you a preview of the request for information so that you can decide how to proceed.

This is used mostly to handle any pagination or batching that needs to take place in order to avoid limits (on either side of the information exchange).

The purpose of planning is to figure out if batches will be needed, how many batches would be needed, and what information is needed by each batch to do its job.

Valence will call planFetch() and hand your Adapter a *LinkContext* instance. Two values in particular are going to be important to you here:

1. **batchSizeLimit** - the desired number of records returned by each call to fetchRecords().
2. **lastSuccessfulSync** - a timestamp for the last time records were successfully retrieved from your adapter. Use this to estimate (or query and count) how many total records you expect to send to Salesforce.

Use these two values, plus what you know about the external system you are connecting to, to plan the retrieval of records.

Planning Result: a FetchStrategy

There are a *number of strategies* you can use to most effectively retrieve records from the system your adapter is talking to.

The return value for planFetch() lets Valence know how best to interact with your Adapter to get the records we're interested in. You have tremendous flexibility and control over this process.

Warning: In between execution contexts your Adapter instance is destroyed and any state is lost. If you want to preserve any information (api tokens, offsets, etc), put it inside scope objects that you can give to Valence to hand back to you later.

2. Fetching

After planning is complete, it is time to retrieve records from the external system that your Adapter talks to.

Depending on the FetchStrategy that you returned from planFetch(), different things may happen here. Read up on the *FetchStrategy* class to understand your options. Also take a look at *ChainFetchAdapter*.

Your fetchRecords() method is invoked with two arguments:

1. **context** - the *LinkContext* instance that contains general information you might need about the current sync that is in progress.
2. **scope** - A scope object you gave Valence to give back to you later, or null, depending on the circumstances. Use these to craft a request to the external system. Perhaps you stored offsets in your scopes so that you could paginate a large result set.

Note: Each time fetchRecords() is invoked it is called on a new instance of your Adapter class that is inside its own execution context. This means you have a fresh set of Salesforce Governor Limits to work within. To see the limits, look at the asynchronous limits here:

https://developer.salesforce.com/docs/atlas.en-us.apexcode.meta/apexcode/apex_gov_limits.htm

Building Your Query

To decide what your query to the external system should be you can look at:

1. **context.linkSourceName** - the name of the source table that should be queried (this will match one of the *Table* names your Adapter returned in *getTables()*).
2. **context.suggestedQueryFields** - a filtered list of fields that Valence thinks should be queried for, based on the user-configured mappings and excluding mappings against injected fields (such as those created by Filters).
3. Any details you gave yourself to work with in the scope (such as page number, page size, offset, etc)

Listing 20: Example

```
List<String> fieldsToSelect = new List<String>();
for(valence.FieldPath fieldPath : context.suggestedQueryFields) {
        fieldsToSelect.add(String.join(fieldPath.getSimplePath(),'.'));
}

String query = String.format('SELECT {0} FROM {1} OFFSET {2} LIMIT {3}', new List<Object>
↪{
        String.join(fieldsToSelect,','),
        context.linkSourceName,
        myScope.offset,
        myScope.limit
});
```

5.48 SourceAdapterForRawDataPush

This interface allows your *Adapter* to be the start of a Push Link that begins with some arbitrary data, in the form of a String.

You will use this interface if your Adapter is processing messages that are being pushed into Salesforce, for example a web hook message. Valence doesn't understand the contents of the message, and passes it to your source Adapter to break it down into RecordInFlight instances.

5.48.1 Definition

```
/**
 * Implement this interface if your Adapter can be used to turn raw source data into␣
↪RecordInFlight records. This means
 * that there is some kind of data you've built your adapter to parse (like JSON in a␣
↪webhook, say). Implementing this
 * interface will allow us to feed that data to your Adapter. Use this interface if your␣
↪Adapter is handed data (a Push).
 */
global interface SourceAdapterForRawDataPush extends SourceAdapter {

        /**
         * Given some source data, produce a collection of RecordInFlight records that␣
↪are ready for further
         * processing by the Valence engine.
         *
         * @param context Information about this Link and the current execution of it.
         * @param rawData Source data as a String; likely holds more than one record.
         *
         * @return The passed source data converted into RecordInFlight records.
         */
        List<RecordInFlight> buildRecords(LinkContext context, String rawData);
}
```

5.49 SourceAdapterForSObjectPush

This interface allows your *Adapter* to be the source Adapter to be part of a Link run that starts out by being given SObject instances.

It is also required if you are using the **LOCATOR** *FetchStrategy*.

Valence's LocalSalesforceAdapter already implements this interface and covers both of the above scenarios, so you would only ever need to implement this if you were writing your own Adapter to interact with the local org for some reason.

5.49.1 Definition

```
/**
 * Implement this interface if your Adapter can be used to turn SObject records into␣
↪RecordInFlight records. In other words,
 * if your Adapter can be used to pick up standard or custom objects (or platform␣
↪events) from the local org and prepare
 * them to be sent elsewhere. Use this interface if your Adapter is handed data␣
↪(sometimes called a Push model).
 */
global interface SourceAdapterForSObjectPush extends SourceAdapter {

        /**
         * Given a collection of sObject records, produce a collection of RecordInFlight␣
```

(continues on next page)

(continued from previous page)

```
↪records that are ready for further
         * processing by the Valence engine.
         *
         * @param context Information about this Link and the current execution of it.
         * @param records Source data, in the form of sObject records. Could be standard/
↪custom objects, or platform events.
         *
         * @return The passed source data converted into RecordInFlight records.
         */
        List<RecordInFlight> buildRecords(LinkContext context, List<SObject> records);
}
```

5.49.2 Example Usage

```
public class MyAdapter implements valence.SourceAdapterForSObjectPush {

    public List<valence.RecordInFlight> buildRecords(valence.LinkContext context, List
↪<sObject> sObjects) {
        List<valence.RecordInFlight> records = new List<valence.RecordInFlight>();
        for(SObject obj : sObjects) {
            Map<String, Object> propertiesMap = obj.getPopulatedFieldsAsMap();
            valence.RecordInFlight record = new valence.RecordInFlight(propertiesMap);
            records.add(record);
        }
        return records;
    }
}
```

5.50 SourceAdapterScopeSerializer

This interface allows Valence to take scopes from your custom *SourceAdapterForPull* Adapter and place them in temporary storage for later retrieval.

By implementing this interface you unlock several powerful diagnostic and recovery features for end users. For example, with serialized batches if a few batches fail outright during a run (say, they have a read timeout calling an external endpoint), then just those batches can be replayed by using the serialized scopes.

You don't have to implement this interface. If you do not, retries will still work but not at the batch level; only for individual failed records.

Tip: Valence encrypts serialized scopes at-rest. They are not stored in plain text.

5.50.1 Definition

```
/**
 * Implementing this interface allows Valence to serialize scopes used to fetch records␣
↪with your Adapter.
 *
 * This allows some powerful behaviors for end users, such as replaying failed batches␣
↪and resuming aborted runs.
 */
global interface SourceAdapterScopeSerializer extends SourceAdapterForPull {

        /**
         * Turn an instance of one of your scope objects into a representation that can␣
↪be easily stored in a database.
         *
         * Valence does NOT store this scope in plaintext.
         *
         * @param scope The scope to serialize
         *
         * @return A persistable representation of your scope instance
         */
        String serializeScope(Object scope);

        /**
         * Take a serialized representation of your object and rehydrate it, creating an␣
↪actual instance of your scope again.
         *
         * @param serializedScope A serialized representation of a scope that needs to␣
↪be deserialized
         *
         * @return A rehydrated instance of your scope class
         */
        Object deserializeScope(String serializedScope);
}
```

5.50.2 Example Usage

Your implementation really never has to get fancier than what is below. You will be asked to turn your scope instances into strings, and then you will be asked later in time to turn those same string values back into scope instances.

```
public String serializeScope(Object scope) {
    return JSON.serialize(scope);
}

public Object deserializeScope(String serializedScope) {
    return JSON.deserialize(serializedScope, MyScopeClass.class);
}

private MyScopeClass {
    Integer pageNumber;
}
```

5.51 TargetAdapter

Implement this interface to allow your *Adapter* to receive records from a Valence Link. Once implemented, your Adapter will show up in Valence as a possible data target.

You can use getBatchSizeLimit to tell Valence the maximum number of records your Adapter can accept at once. Valence will make sure batches are broken down to this size.

Warning: It is a **requirement** that your target Adapter respects the Link setting for *Testing Mode*, which an admin user sets when they don't want records to actually persist into the target system.

You will know if a Link run is in testing mode if the *LinkContext* `testingMode` boolean property is set to true. At a minimum, you can simply immediately return from pushRecords():

```
public void pushRecords(valence.LinkContext context, List<valence.RecordInFlight>␣
↪records) {

        // don't send any data if this Link is running in testing mode
        if(context.testingMode == true) {
                return;
        }
```

If possible, it's nice if you have a mechanism to do a trial push where you can test a write but roll back so that you can attach any errors or warnings to the RecordInFlight instances. Very few APIs support a mechanic like this, so we don't expect it but it's a nice to have.

5.51.1 Definition

```
/**
 * Implement this interface if your Adapter can receive records from Valence.
 */
global interface TargetAdapter extends Adapter {

        /**
         * Specify a limit for how many records your endpoint can receive in a single␣
↪batch.
         *
         * @param context Information about this Link and the current execution of it.
         *
         * @return An upper bound on how many records Valence should put in each batch␣
↪when sending records to your adapter.
         */
        Integer getBatchSizeLimit(LinkContext context);

        /**
         * Send records to the adapter. The size of the List will not exceed the value␣
↪returned by getBatchSizeLimit().
         *
         * @param context Information about this Link and the current execution of it.
         * @param records Records that have been received from another system, processed,
↪ and are ready for delivery.
         */
```

(continues on next page)

(continued from previous page)

```
        void pushRecords(LinkContext context, List<RecordInFlight> records);
}
```

5.52 ConfigurablePerLinkFilter

This interface allows a *Filter* to have multiple configurations applied to it on a given Link that are independent of any particular *Mapping*. If you need configurations that are different for a given Mapping, look at *ConfigurablePerMappingFilter*.

Users can set up multiple configurations for your Filter on a single Link, so you'll notice that `setFilterConfigurations()` accepts a collection of serialized configurations and you should be prepared to apply each one to records.

Each actual String **configuration** value is whatever you need it to be. Most commonly it is a serialized JSON object of key-value pairs, but you are not locked into using that pattern.

Note: For a deeper look at configurable extensions in Valence, read our guide on *Configurability*.

5.52.1 Definition

```
/**
 * Implementing this interface means your Filter is capable of being configured.
 */
global interface ConfigurablePerLinkFilter extends TransformationFilter {

        /**
         * You can use your own Lightning component to let Users build and edit your␣
↪configuration. If you want to do this, return the fully qualified
         * name of your component, which looks like this:
         *
         * valence:MyAwesomeAdapterConfigurator
         *
         * Make sure your component is set to global so that Valence can instantiate it.
         *
         * @return The name of your Lightning component that will handle configuration,␣
↪or null if you don't need your own component
         */
        String getFilterConfigurationLightningComponent();

        /**
         * If you don't need or don't want to use your own Lightning Component, you can␣
↪simply describe your configuration shape and we will present
         * the user with some basic input fields to populate values in your␣
↪configuration.
         *
         * @return A serialized JSON object describing your configuration data structure,
↪ or null if you use your own component
         */
```

(continues on next page)

(continued from previous page)

```
        String getFilterConfigurationStructure();

        /**
         * Given configuration data, return a user-friendly paragraph that explains how␣
↪this specific configuration
         * is going to be used by your Filter and what effect that will have on the␣
↪Record.
         *
         * We show this in the user interface to help Users understand the impact of␣
↪their configurations.
         *
         * @param configurationData The raw configuration for your Filter
         *
         * @return A human-readable and friendly explanation that specifically reflects␣
↪and explains the configuration passed.
         */
        String explainFilterConfiguration(String configurationData);

        /**
         * Sets configuration data for your Filter. This is the first method called on␣
↪your Filter during Link execution. User might configure your Filter multiple
         * times (for example, setting two different constants for a Filter that sets␣
↪constants), so you are passed a list of configurations.
         *
         * @param context Information about this Link and the current execution of it.
         * @param configurationData List of configuration data in JSON format, or in␣
↪whatever format your custom configuration component gave us
         */
        void setFilterConfigurations(LinkContext context, List<String>␣
↪configurationData);
}
```

5.52.2 Example - Access Configurations

```
private List<FilterConfiguration> configs = new List<FilterConfiguration>();

public void setFilterConfigurations(valence.LinkContext context, List<String>␣
↪configurationData) {
        configs.clear();
        for(String configData : configurationData) {
                configs.add(interpretConfigData(configData));
        }
}

public void process(valence.LinkContext context, List<valence.RecordInFlight> records) {

        // do work on the records using our configurations
        for(valence.RecordInFlight record : records) {
                for(FilterConfiguration config : configs) {
                        record.setOriginalPropertyValue(config.fieldName, config.value);
```

(continues on next page)

(continued from previous page)

```
                }
        }
}

private static FilterConfiguration interpretConfigData(String data) {
        return String.isBlank(data) ? new FilterConfiguration() :␣
↪(FilterConfiguration)JSON.deserialize(data, FilterConfiguration.class);
}

private class FilterConfiguration {
        String fieldName;
        String value;
}
```

5.53 ConfigurablePerMappingFilter

This interface allows a *Filter* to have a configuration applied to it that is different for each *Mapping* on the Link. This allows an admin to make the Filter behave differently for each Mapping, or skip certain Mappings. If you need configurations that aren't tied to a particular Mapping, look at *ConfigurablePerLinkFilter*.

This interface follows our normal pattern for a configurable extension in Valence, with a slight variation: instead of a setConfiguration method, you can find the configuration for each mapping on the Mapping itself that you get inside a *LinkContext*. The LinkContext is tailored to each Filter that is called, so the mappings will have configurations for just your Filter, or nothing. See an example of how to access these configurations below.

The actual String **configuration** value is whatever you need it to be. Most commonly it is a serialized JSON object of key-value pairs, but you are not locked into using that pattern.

Note: For a deeper look at configurable extensions in Valence, read our guide on *Configurability*.

5.53.1 Definition

```
/**
 * Implementing this interface means your Filter is capable of being configured per-
↪Mapping. The configurations
 * for the mappings will be fed to your Filter when it receives RecordInFlight records␣
↪to process.
 */
global interface ConfigurablePerMappingFilter extends TransformationFilter {

    /**
     * You can use your own Lightning component to let Users build and edit your␣
↪configuration. If you want to do this, return the fully qualified
     * name of your component, which looks like this:
     *
     * valence:MyAwesomeAdapterConfigurator
     *
     * Make sure your component is set to global so that Valence can instantiate it.
```

(continues on next page)

(continued from previous page)

```
     *
     * @return The name of your Lightning component that will handle configuration, or␣
↪null if you don't need your own component
     */
    String getMappingConfigurationLightningComponent();

    /**
     * If you don't need or don't want to use your own Lightning Component, you can␣
↪simply describe your configuration shape and we will present
     * the user with some basic input fields to populate values in your configuration.
     *
     * @return A serialized JSON object describing your configuration data structure, or␣
↪null if you use your own component
     */
    String getMappingConfigurationStructure();

    /**
     * Given mapping configuration data, return a user-friendly paragraph that explains␣
↪how this specific configuration
     * is going to be used by your Filter and what effect that will have on the Record.
     *
     * We show this in the user interface to help Users understand the impact of their␣
↪configurations.
     *
     * @param configuration The raw configuration for a specific mapping
     *
     * @return A human-readable and friendly explanation that specifically reflects and␣
↪explains the configuration passed.
     */
    String explainMappingConfiguration(String configuration);
}
```

5.53.2 Example - Access Configurations

```
public void process(valence.LinkContext context, List<valence.RecordInFlight> records) {

    // extract mappings from the context object
    List<valence.Mapping> mappings = context.mappings.values();

    // look for configurations
    for(valence.Mapping mapping : mappings) {
        if(String.isNotBlank(mapping.configuration)) {
            // this mapping has a configuration for your filter
        }
    }
}
```

5.54 LinkSplitFilter

This interface allows a *Filter* to be part of the logic flow of one or more Link Splits.

LinkSplitFilters are not extensions of *TransformationFilter* and are not expected to mutate RecordInFlight instances.

Instead, their job is to evaluate each RecordInFlight and decide if it should, or should not, be sent to a given Link Split child of a running Link. They are, in some ways, a router directing traffic.

5.54.1 Definition

```
/**
 * A Filter that is used to evaluate RecordInFlight instances to see if they should be␣
↪processed by a Link Split.
 */
global interface LinkSplitFilter {

        /**
         * Consider a List of RecordInFlight instances against the LinkContext they␣
↪would be run against if they were allowed to proceed.
         *
         * @param context Details about the Link that would run if some records are␣
↪accepted
         * @param records The records to evaluate
         *
         * @return An ordered list of true/false, where each item is the acceptance/
↪rejection of the corresponding index in the records parameter
         */
        List<Boolean> evaluate(LinkContext context, List<RecordInFlight> records);
}
```

5.54.2 Example

Let's say you have a Link called "AllRecords", whose job it is to pick up any records that have changed in a remote database regardless of the table they come from.

You are expecting three flavors of records:

1. Companies (which you want to put in Account)
2. People (which you want to put in Contact)
3. Invoices (which you want to put in Invoice__c)

Consequently, you have three additional Links (one for each of these types of records).

AllRecords runs, and it has 20 records that are a mix of all three types. How do you decide what to do next? That's where LinkSplitFilter comes in.

If the User selects a LinkSplitFilter when they are setting up the three Link Splits (one for each of our data types) that Filter will get a chance to give a thumbs up or down on each record.

The `evaluate()` method will be invoked three times, with a different LinkContext each time (corresponding to each of our child Links). The logic below would sort records appropriately.

```
public List<Boolean> evaluate(valence.LinkContext context, List<valence.RecordInFlight>␣
↪records) {

            List<Boolean> results = new List<Boolean>();

    for(valence.RecordInFlight record : records) {

                    // let's say that the AllRecords Link adds a field to each record␣
↪called "dataType" that holds a string value indicating what flavor of record it is
                    String dataType = (String)record.getOriginalPropertyValue('dataType
↪');

                    switch on dataType {
                            when 'company' {
                                    results.add('Account'.equalsIgnoreCase(context.
↪linkTargetName));
                            }
                            when 'person' {
                                    results.add('Contact'.equalsIgnoreCase(context.
↪linkTargetName));
                            }
                            when 'invoice' {
                                    results.add('Invoice__c'.equalsIgnoreCase(context.
↪linkTargetName));
                            }
                    }
            }

            return results;
}
```

5.55 SchemaAwareTransformationFilter

This interface allows a *Filter* to describe their impact on record schema to Valence, much like an Adapter can with *SchemaAdapter*.

This is a nice-to-have that we **strongly** encourage you to implement, as it makes life much better for Users. Implementing this allows them to understand which fields they see on the records came from which Filters, and why.

To learn more about how Valence uses schema information, check out our *Schema* page.

5.55.1 Definition

```
/**
 * A "schema-aware" TransformationFilter is able to describe its affect on the records␣
↪that it interacts with. It
 * explains how it might modify the record, what fields it depends on, what fields it␣
↪adds, etc.
 */
global interface SchemaAwareTransformationFilter extends TransformationFilter {
```

(continues on next page)

(continued from previous page)

```
        /**
         * Help us understand how your Filter interacts with Records. Given a specific␣
↪LinkContext (with included Mappings and configurations),
         * describe what source and target fields you would read, create, change, and␣
↪write to.
         *
         * @param context Details about a potential Link run
         *
         * @return An instance of FilterSchema, describing fields this Filter reads,␣
↪changes, creates, writes to, etc
         */
        FilterSchema describeSchema(LinkContext context);
}
```

5.55.2 Supporting Classes

You can see above that the return type is an instance of **FilterSchema**, which is a convenience class that makes it easier to describe what your Filter does.

Below are the signatures from that class.

```
// constructor
global FilterSchema();

global FilterSchema addTouch(Touch newTouch);

global void createsSourceField(Field newField);

global static Touch buildSourceTouch(String operation);

global static Touch buildTargetTouch(String operation);

// inner class valence.FilterSchema.Touch
global class Touch {

        global Touch onField(Mapping mapping); // give credit to this mapping and attach␣
↪to the source or target side of it (depending on how you built it)

        global Touch onField(List<String> path); // attach to a source or target field␣
↪independent of a mapping

        global Touch creditMapping(Mapping mapping); // give credit to a mapping for a␣
↪field that is not actually the source or target of that mapping (derivative)
}
```

5.55.3 Example - Always Read Same Field

In this simple example a Filter depends on a specific field that it always reads from, regardless of context.

```
valence.FilterSchema describeSchema(valence.LinkContext context) {
        // always reads 'flavor' field no matter what the Link is or what Mappings exist␣
↪on it
        return new valence.FilterSchema()
                .addTouch(valence.FilterSchema.buildSourceTouch('READ').onField(new List
↪<String>{'flavor'}));
}
```

5.55.4 Example - Write To a Field That Isn't Part of A Mapping

Here's a slightly more complex example. Let's say your Filter is configured per-mapping, and what it does is create an additional field on the target side in addition to the target field the mapping is normally going to write to (this is exactly how our native RelationshipFilter works).

```
schema.addTouch(FilterSchema.buildSourceTouch('READ').onField(mapping)); // reads the␣
↪source side of the mapping
schema.addTouch(FilterSchema.buildTargetTouch('WRITE').onField(new List<String>{config.
↪targetFieldName}).creditMapping(mapping)); // write to a separately-configured field␣
↪but give credit to this mapping for why you wrote there
```

5.55.5 Example - Create Fields Based on Configurations

There's some additional special things that happen when your Filter uses `createsSourceField()`. This will make your Field appear in the schema browser for Users, and they can actually build mappings and other configurations against it because they know to expect it. The fields you create can still be dynamic though and based on some configuration you have been set up with.

Note: Any Field you declare with `createsSourceField()` will automatically have a "WRITE" Touch added to it.

```
/*
* Let's say this Filter implements ConfigurablePerLinkFilter and had some configurations␣
↪set that hold fieldNames and values for writing constants.
*/

private List<FilterConfiguration> configs = new List<FilterConfiguration>();

public void setFilterConfigurations(valence.LinkContext context, List<String>␣
↪configurationData) {
        configs.clear();
        for(String configData : configurationData) {
                configs.add(interpretConfigData(configData));
        }
}

public void process(valence.LinkContext context, List<valence.RecordInFlight> records) {
```

(continues on next page)

(continued from previous page)

```
        // do work on the records using our configurations
        for(valence.RecordInFlight record : records) {
                for(FilterConfiguration config : configs) {
                        record.setOriginalPropertyValue(config.fieldName, config.value);
                }
        }
}

private static FilterConfiguration interpretConfigData(String data) {
        return String.isBlank(data) ? new FilterConfiguration() :␣
↪(FilterConfiguration)JSON.deserialize(data, FilterConfiguration.class);
}

private class FilterConfiguration {
        String fieldName;
        String value;
}

valence.FilterSchema describeSchema(valence.LinkContext context) {

        valence.FilterSchema schema = new valence.FilterSchema();

        for(FilterConfiguration config : configs) {
                schema.createsSourceField(valence.Field.create(config.fieldName)
                .withDescription(String.format('A static field created by the Constants␣
↪Filter whose value will always be <strong>{0}</strong>', new List<Object>{config.value}
↪)).build());
        }
        return schema;
}
```

Tip: This last example comes from a real Filter you can go look at on GitHub.

5.56 TransformationFilter

This is the primary interface to implement if you are building a *Filter*. It will allow your Apex class to inspect records that are being processed, and modify them if need be.

Filters are given an opportunity to inspect and modify *RecordInFlight* instances during Link processing, after they are retrieved/handled by a Source Adapter but before they are delivered to a *TargetAdapter*.

5.56.1 Definition

```
/**
 * TransformationFilters are applied to RecordInFlight records.
 */
global interface TransformationFilter {

        /**
         * Given contextual information about a Link, let us know if this Filter is␣
↪appropriate to use in this context. Perhaps your link only works
         * in certain scenarios, like when the local Salesforce org is the source system.
         *
         * If you're not sure what to do with this method, just return true.
         *
         * @param context Information about this Link execution in case you need it
         *
         * @return True if this Filter is appropriate to use with this Link
         */
        Boolean validFor(LinkContext context);

        /**
         * This method allows the Filter to inspect and possibly manipulate␣
↪RecordInFlight records. These records represent a record that is
         * being processed inside of a Valence Link and is in the midst of syncing from␣
↪a source system to a target system.
         *
         * Change records in the list, or the list itself, to affect what ends up in the␣
↪target system. You can add/remove/change properties,
         * add errors or warnings to the record, or even remove it from the list␣
↪entirely to drop it on the floor without causing a failed record.
         *
         * Word of warning: Do not add new records to the list. We've carefully␣
↪calculated list size in order to not hit governor limits, and also
         * any new records are not going to be processed by filters that were applied␣
↪before yours was.
         *
         * @param context Information about this Link execution in case you need it
         * @param records A subset (single batch) of records being processed by the Link
         */
        void process(LinkContext context, List<RecordInFlight> records);
}
```

5.56.2 Example Usage

```
global with sharing class ConstantFilter implements valence.TransformationFilter {

    public Boolean validFor(valence.LinkContext context) {
        // allow this filter to be used by any Link in any circumstance
        return true;
    }

```

(continues on next page)

(continued from previous page)

```
    public void process(valence.LinkContext context, List<valence.RecordInFlight>␣
↪records) {
        // add a constant property to every record that goes by
        for(valence.RecordInFlight record : records)
            record.getProperties().put('SpecialField', 'HelloThere');
    }
}
```