

---

# Valence

Mar 06, 2021



<b>1</b>	<b>Concepts</b>	<b>3</b>
<b>2</b>	<b>Guides</b>	<b>5</b>
2.1	Overview . . . . .	5
2.2	Configuration Files . . . . .	6
2.3	External Systems . . . . .	8
2.4	Custom Extensions . . . . .	9
2.5	Sending Realtime Records To Salesforce . . . . .	15
2.6	Using Dynamic Configurations in your Extension . . . . .	16
2.7	How to Design a Complementary Valence API . . . . .	20
2.8	Building a Configurable Filter that Ignores Records Based on User-selected Cutoff Date . . . . .	22
2.9	Using Valence Between Two Salesforce Orgs (aka Salesforce to Salesforce) . . . . .	27
2.10	AdapterException . . . . .	34
2.11	FetchStrategy . . . . .	35
2.12	Field . . . . .	40
2.13	FilterException . . . . .	41
2.14	LinkContext . . . . .	42
2.15	Mapping . . . . .	43
2.16	RecordInFlight . . . . .	44
2.17	Table . . . . .	46
2.18	ChainFetchAdapter . . . . .	46
2.19	ConfigurableSourceAdapter . . . . .	48
2.20	ConfigurableTargetAdapter . . . . .	49
2.21	DelayedPlanningAdapter . . . . .	52
2.22	NamedCredentialAdapter . . . . .	53
2.23	SchemaAdapter . . . . .	55
2.24	SourceAdapterForPull . . . . .	56
2.25	SourceAdapterForRawDataPush . . . . .	58
2.26	SourceAdapterForSObjectPush . . . . .	59
2.27	SourceAdapterScopeSerializer . . . . .	60
2.28	TargetAdapter . . . . .	61
2.29	ConfigurablePerMappingFilter . . . . .	62
2.30	TransformationFilter . . . . .	64



Valence is a tool designed to make your life easier. 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 code is needed.



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

- *Overview* — Valence is a middleware engine that lives inside your Salesforce org. Data movement is handled with direct connections between Salesforce and external systems. There are several building block concepts that are important to understand in order to configure Valence.
- *Configuration Files* — Valence configurations are stored as Custom Metadata Type records, and can easily be moved between Salesforce environments.
- *External Systems* — Valence reaches out using HTTP to exchange data with external systems. External systems can also call into Valence using the Apex REST API.
- *Custom Extensions* — 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.





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

- *Sending Realtime Records To Salesforce*
- *Using Dynamic Configurations in your Extension*
- *How to Design a Complementary Valence API*
- *Building a Configurable Filter that Ignores Records Based on User-selected Cutoff Date*
- *Using Valence Between Two Salesforce Orgs (aka Salesforce to Salesforce)*

## 2.1 Overview

Valence is a middleware engine that lives inside your Salesforce org. Data movement is handled with direct connections between Salesforce and external systems. There are several building block concepts that are important to understand in order to configure Valence.

### 2.1.1 Links

Building Links is the main mechanism by which you set up Valence to move data between systems. 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.

Links can share access credentials (a NamedCredential), Adapters, and Filters. Each Link has its own mappings, schedule, settings, configurations and analytics.

There are two flavors of Links:

1. **Pull** - Valence fires at scheduled intervals and asks the source system for data, then sends that data to the target system.

2. **Push** - Event-driven rather than scheduled, the source system notifies Valence of new or changed data and Valence processes that data and delivers it to the target system.

Salesforce can be the source or target system (or should you choose, both) for both Pull and Push Links.

### 2.1.2 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.

### 2.1.3 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.

## 2.2 Configuration 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, we have included 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.

## 2.2.1 Valence Custom Metadata Types

API Name	Label	Description
va- lence__ValenceAdapter	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.
va- lence__ValenceDataLink	Valence Data Link	Encapsulates the idea of two tables in different systems being linked to each other, and having records move between those tables.
va- lence__ValenceDataLink	Valence Data Link Filter	Junction table that connects Links to Filters.
va- lence__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.
va- lence__ValenceMapping	Valence Map- ping	A mapping links a field in an external system to a field in this Salesforce organization.
va- lence__ValenceMapping	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 *Custom Extensions*, you will be manually creating Valence Adapter and Valence Filter records to represent your Apex classes.

## 2.2.2 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		
<input type="checkbox"/> arFka9	Accounts	Link	✓	Select With Descendants	Deselect With Descendants
<input type="checkbox"/> a3TIBK	Key Filter	Link Filter	✓	Select With Descendants	Deselect With Descendants
<input type="checkbox"/> aOaEdm	Relationship Filter	Link Filter	✓	Select With Descendants	Deselect With Descendants
<input type="checkbox"/> ayHFZ5	Id (Company Id) => Sic (SIC Code)	Mapping	✓	Select With Descendants	Deselect With Descendants
<input checked="" type="checkbox"/> aJyQOU	Parent (Parent Company) => Pare...	Mapping	✓	Select With Descendants	Deselect With Descendants
<input checked="" type="checkbox"/> a9nmaV	Relationship Filter	Filter Configuration	✓	Select With Descendants	Deselect With Descendants
<input type="checkbox"/> aghRAI	Phone (Phone Number) => Phon...	Mapping	✓	Select With Descendants	Deselect With Descendants
<input checked="" type="checkbox"/> 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.
- 

## 2.3 External Systems

Valence reaches out using HTTP to exchange data with external systems. External systems can also call into Valence using the Apex REST API.

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.

Ideally, someone has already written an Adapter for the specific external system you are interested in connecting to. Search the AppExchange and Github for Valence extensions, or reach out to ask us if we're aware of specific Adapters. Eventually we will expose an Adapter indexing service and a way to search and install them.

### 2.3.1 Authentication

Valence uses the native Salesforce [Named Credential](#) feature to handle:

1. Defining external systems that can be interacted 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.

Out of the box, Named Credential supports two authentication methods:

- Standard username + password header authentication.
- OAuth using the "authorization code" grant type.

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.

### 2.3.2 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*.

---

## 2.4 Custom Extensions

### 2.4.1 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 *custom UI* 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.

### 2.4.2 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
<i>AdapterException</i>	Special Exception class that we encourage you to use in your Adapters to indicate that something has gone wrong that cannot be recovered from.
<i>FetchStrategy</i>	FetchStrategy allows your source Adapter to tell Valence how best to ask your Adapter for records. It is an example of the Strategy Pattern <a href="https://en.wikipedia.org/wiki/Strategy_pattern">https://en.wikipedia.org/wiki/Strategy_pattern</a> .
<i>Field</i>	The Field class represents a property that a record may have. It is analogous to a table column, or in Salesforce an object field.
<i>FilterException</i>	Special Exception class that we encourage you to use in your Filters to indicate that something has gone wrong that cannot be recovered from.
<i>LinkContext</i>	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.
<i>Mapping</i>	Mapping is a special Apex class that gives Adapters and Filters info about the mappings a user has defined for the Link.
<i>RecordInFlight</i>	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.
<i>Table</i>	The Table class represents a possible source or target for a Link. It is analogous to a database table or Salesforce object.

## Adapter Interfaces

Inter- face	Link Model	Source/Target	Description
<i>Chain- FetchAdapter</i>	Pull	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.
<i>Config- urable- SourceAdapter</i>	Ei- ther	Source	Implement if your Adapter is user-configurable when used as the <b>source</b> of records on a Link.
<i>Config- urable- Targe- tAdapter</i>	Ei- ther	Tar- get	Implement if your Adapter is user-configurable when used as the <b>target</b> of records on a Link.
<i>Named- Cre- dential- Adapter</i>	Ei- ther	Ei- ther	Implement if your Adapter supports using a NamedCredential as its source of endpoint and credential info. Almost every Adapter will likely implement this interface.
<i>SchemaAdapter</i>	Ei- ther	Ei- ther	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.
<i>SourceAdapter ForPull</i>	Pull	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.
<i>SourceAdapter ForRaw- Data- Push</i>	Push	Source	Implement if your Adapter is parsing raw data (like a JSON packet) as the beginning of a <b>Push</b> model Link. You'd use this if you had real-time data packets arriving, for example via API into the Salesforce org.
<i>SourceAdapter ForSOB- jectPush</i>	Push	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.
<i>SourceAdapter ScopeSe- rializer</i>	Pull	Source	Implement so that users can replay failed batches on Links where your Adapter was the source.
<i>Targe- tAdapter</i>	Ei- ther	Tar- get	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
<i>Config- urablePerMap- pingFilter</i>	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.
<i>Transforma- tionFilter</i>	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.

### 2.4.3 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** [Edit](#) [Delete](#) [Clone](#)

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	<input type="checkbox"/>
Is Test	<input type="checkbox"/>	Requires NamedCredential for Data	<input type="checkbox"/>

▼ System Fields

Protected Component	<input type="checkbox"/>	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
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 Adapter class that will be instantiated.
Namespace__c	Namespace	The namespace of the Adapter class that will be instantiated. <b>Does not</b> 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 NamedCredential for Schema	Checked if this Adapter must be given access to a NamedCredential in order to return its Schema.
Requires-NamedCredentialForData__c	Requires NamedCredential 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 <b>Protected Component</b> field as well.

## Register a Custom Filter

Telling Valence about your custom Adapter is done by making a new record in the **valence\_\_ValenceFilter\_\_mdt** custom metadata type.

**Valence Filter Detail** Edit Delete Clone

Label	Key Filter	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	<input checked="" type="checkbox"/>
			Default Sort Order	1.0000

---

▼ System Fields

Protected Component	<input type="checkbox"/>	Created By	User User, 1/24/2018 5:15 PM
Namespace Prefix	valence	Last Modified By	User User, 1/24/2018 5:15 PM

Edit Delete Clone

API Name	Label	Description
Developer-Name	Name	The API name of this record, used whenever this record is edited or retrieved programmatically.
Master-Label	Label	The user-friendly display label for this record.
Class-Name__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. <b>Does not</b> 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.

## 2.4.4 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.

---

## 2.5 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.

To do real-time behavior in Valence we use a Push Link, which is a flavor of Link that expects records to be pushed from the source system (as opposed to querying for them at scheduled intervals).

Push Links are surfaced using the [Salesforce Apex REST API](#). As an external system you would authenticate with Salesforce and then send messages to an Apex REST endpoint.

### 2.5.1 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](https://developer.salesforce.com/docs/atlas.en-us.api_rest.meta/api_rest/intro_understanding_authentication.htm)

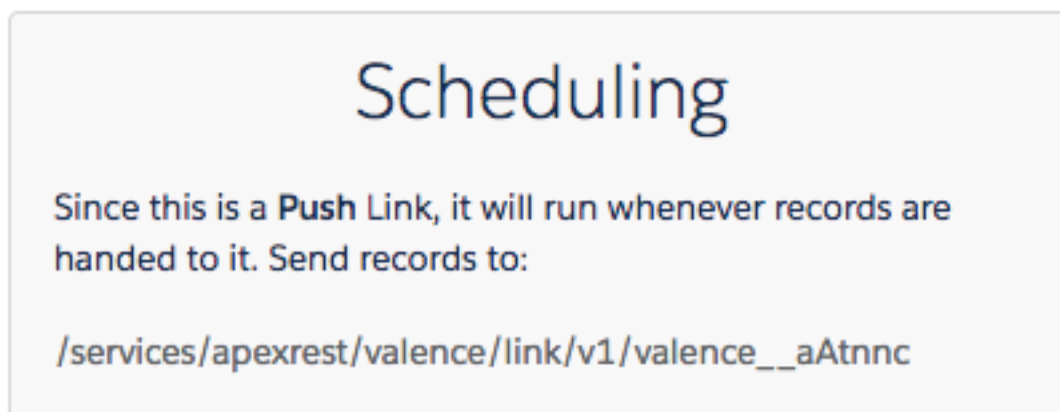
---

**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.

---

### 2.5.2 Sending The Message

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



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. 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..."
  }
]
```

## 2.6 Using Dynamic Configurations in your Extension

In certain places in Valence we allow you to define a configuration schema that Valence parses and uses to render a UI for the user to do the configuration.

Typically, you have three levels of configurability available to your custom adapters and filters, listed below from least sophisticated to most sophisticated:

1. No configuration necessary.
2. *Configuration schema* defined, automatically turned into a UI by Valence with the user-defined values stored and sent back to your extension when Links are running.
3. *Custom UI component* delivered by your extension and embedded inside the Valence app; configuration obtained from your UI by Valence, stored and sent back to your extension when Links are running.

By default extensions are not configurable. You can change this (and indicate that your extension can be configured) by implementing one of these interfaces:

Adapter interfaces:

- *ConfigurableSourceAdapter*
- *ConfigurableTargetAdapter*

Filter interfaces:

- *ConfigurablePerMappingFilter*

---

**Note:** No matter which configuration option you pick, you do not have to manage any kind of interaction with the data store. Your configuration data will be passed back and forth between you and the Valence framework, and Valence will take care of serializing configurations and giving them back to you again later at the appropriate time.

---

## 2.6.1 Configuration Schema

Configuration Schema allows you to define an expected shape of your configuration and Valence will handle creating the UI and getting values from users.

**Warning:** **Configuration Schema** and **Custom UI Component** are mutually exclusive. Either you are responsible for the UI, or Valence is. If you go with Configuration Schema you will return a value from `getConfigurationStructure()` and return **null** from `getConfigurationLightningComponent()`.

Valence will expect a serialized JSON string to be returned by `getConfigurationStructure()`. The framework uses this structure to dynamically instantiate a form for the user to fill out.

### 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.

#### 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.

## Configuration Schema 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
      }
    }
  ]
}
```

## 2.6.2 Custom UI Component

If you would like to use your own custom Lightning component to render a user experience to configure your extension, return the fully qualified name of your component from the `getConfigurationLightningComponent()` method.

**Warning:** **Configuration Schema** and **Custom UI Component** are mutually exclusive. Either you are responsible for the UI, or Valence is. If you go with Custom UI Component you will return a fully qualified Lightning component name from `getConfigurationLightningComponent()` and return **null** from `getConfigurationStructure()`.

## Example

```
public String getConfigurationLightningComponent() {
    return 'yourPackageName:SomeLightningConfigurator';
}
```

Your Lightning component must implement a Lightning Interface called “valence:Configurator”.

## valence:Configurator Interface

```
<aura:interface description="Implementors of this interface can be instantiated in
↳the Valence UI so Users can configure their adapters, filters, etc.">
    <aura:attribute name="link" type="Object" description="Information about the Link
↳that may prove useful during configuration." />
    <aura:attribute name="configuration" type="Object" description="The configuration
↳object. Valence will load and persist this for you." />
    <aura:attribute name="isValid" type="Boolean" description="A flag to indicate if
↳the configuration is allowed to be saved." />
    <aura:attribute name="isDirty" type="Boolean" description="A flag to indicate if
↳the configuration has been modified." />
</aura:interface>
```

All four of these attributes will be passed to your component when it is instantiated. Here’s how they work:

### Link (read-only)

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

### isValid

Use this third attribute to control the user’s ability to save the configuration. As long as it is set to false, the user’s Save button that they see in the top right corner will be disabled. Set this to true whenever the user has done whatever you feel is necessary to get your configuration into a valid state. Don’t hesitate to toggle this on and off as the user makes changes.

### isDirty

Use this fourth attribute to control the user's ability to save the configuration. As long as it is set to false, no action buttons (Save/Discard) will be shown to the User. Set this to true whenever the User makes changes to the configuration that cause it to be different than what they started with. Don't hesitate to toggle this on and off as the user makes changes.

## 2.7 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.

### 2.7.1 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>

## 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 two authentication methods out of the box:

1. Standard username + password header authentication.
2. OAuth using the "authorization code" grant type.

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.

### 2.7.2 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.

## 2.8 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.

### 2.8.1 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.

### 2.8.2 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 Schema* pattern for handling configurations.

#### Class Declaration

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

```

1  /**
2   * Valence filter that allows us to set a date threshold that will cause records to
   * be ignored if a given
3   * field from that record is older than our threshold.
4   */
5  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 *Configuration Schema*, 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.

```

7   public String getMappingConfigurationLightningComponent() {
8       return null;
9   }
10
11  public String getMappingConfigurationStructure() {
12      return JSON.serialize(new Map<String, Object>{
13          'description' => '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).',
14          'fields' => new List <Map<String, Object>>{
15              new Map<String, Object>{
16                  'name' => 'cutoff',
17                  'attributes' => new Map<String, Object>{
18                      'label' => 'Cutoff Date',
19                      'type' => 'date'
20                  }
21              }
22          }
23      });
24  }

```

**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.

```

26     public String explainMappingConfiguration(String configuration) {
27
28         String explanation = 'This Filter will set records to be ignored if their_
↪field value (the one in this mapping) is older than {0}.';
29
30         Configuration config = (Configuration)JSON.deserialize(configuration,
↪IgnoreOldRecordsFilter.Configuration.class);
31
32         return String.format(explanation, new List<String>{String.valueOf(config.
↪cutoff)});
33     }

```

## 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.

```

75     /**
76      * Simple class for holding the configuration needed for this filter.
77      */
78     private class Configuration {
79         private Date cutoff;
80     }

```

## 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 out-bound.

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

```

35     public Boolean validFor(valence.LinkContext context) {
36         return true;
37     }

```

## 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.

```

39     public void process(valence.LinkContext context, List<valence.RecordInFlight>_
↪records) {
40
41         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.

```

46     for(valence.Mapping mapping : context.mappings.values()) {
47
48         // skip blank configurations
49         if(String.isNotBlank(mapping.configuration)) {
50             Configuration config = (Configuration)JSON.deserialize(mapping.
↪configuration, IgnoreOldRecordsFilter.Configuration.class);
51             cutoffsBySourceField.put(mapping.sourceFieldName, config.cutoff);
52         }
53     }

```

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

```

55     // bail out if we didn't find any
56     if(cutoffsBySourceField.isEmpty())
57         return;

```

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

```

62     for(valence.RecordInFlight record : records) {

```

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

```

63         for(String sourceField : cutoffsBySourceField.keySet()) {
64             Date cutoff = cutoffsBySourceField.get(sourceField);
65             Long fieldValue = (Long)record.getOriginalProperties().
↪get(sourceField);

```

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

```

66             if(fieldValue != null) {
67                 if(Datetime.newInstance(fieldValue).dateGmt() < cutoff) {
68                     record.ignore('Record field <' + sourceField + '> older than
↪cutoff of ' + cutoff);
69                 }

```

**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!

## 2.8.3 Full Solution Code

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

```

1  /**
2   * Valence filter that allows us to set a date threshold that will cause records to
↪be ignored if a given
3   * field from that record is older than our threshold.
4   */
5  global with sharing class IgnoreOldRecordsFilter implements valence.
↪TransformationFilter, valence.ConfigurablePerMappingFilter {
6
7     public String getMappingConfigurationLightningComponent() {

```

(continues on next page)

```

8     return null;
9     }
10
11    public String getMappingConfigurationStructure() {
12        return JSON.serialize(new Map<String, Object>{
13            'description' => '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).',
14            'fields' => new List <Map<String, Object>>{
15                new Map<String, Object>{
16                    'name' => 'cutoff',
17                    'attributes' => new Map<String, Object>{
18                        'label' => 'Cutoff Date',
19                        'type' => 'date'
20                    }
21                }
22            }
23        });
24    }
25
26    public String explainMappingConfiguration(String configuration) {
27
28        String explanation = 'This Filter will set records to be ignored if their
↳field value (the one in this mapping) is older than {0}.';
29
30        Configuration config = (Configuration)JSON.deserialize(configuration,
↳IgnoreOldRecordsFilter.Configuration.class);
31
32        return String.format(explanation, new List<String>{String.valueOf(config.
↳cutoff)});
33    }
34
35    public Boolean validFor(valence.LinkContext context) {
36        return true;
37    }
38
39    public void process(valence.LinkContext context, List<valence.RecordInFlight>
↳records) {
40
41        Map<String, Date> cutoffsBySourceField = new Map<String, Date>();
42
43        /*
44         * Assemble any cutoffs that have been configured by admins.
45         */
46        for(valence.Mapping mapping : context.mappings.values()) {
47
48            // skip blank configurations
49            if(String.isNotBlank(mapping.configuration)) {
50                Configuration config = (Configuration)JSON.deserialize(mapping.
↳configuration, IgnoreOldRecordsFilter.Configuration.class);
51                cutoffsBySourceField.put(mapping.sourceFieldName, config.cutoff);
52            }
53        }
54
55        // bail out if we didn't find any
56        if(cutoffsBySourceField.isEmpty())
57            return;

```

(continues on next page)

(continued from previous page)

```

58
59     /*
60     * Iterate through our records, ignoring where appropriate based on cutoff_
↪dates.
61     */
62     for(valence.RecordInFlight record : records) {
63         for(String sourceField : cutoffsBySourceField.keySet()) {
64             Date cutoff = cutoffsBySourceField.get(sourceField);
65             Long fieldValue = (Long)record.getOriginalProperties().
↪get(sourceField);
66             if(fieldValue != null) {
67                 if(Datetime.newInstance(fieldValue).dateGmt() < cutoff) {
68                     record.ignore('Record field <' + sourceField + '> older than_
↪cutoff of ' + cutoff);
69                 }
70             }
71         }
72     }
73 }
74
75 /**
76 * Simple class for holding the configuration needed for this filter.
77 */
78 private class Configuration {
79     private Date cutoff;
80 }
81 }

```

## 2.9 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.

## 2.9.1 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".

The screenshot shows the 'New Connected App' configuration page in Salesforce. The 'Basic Information' section contains the following fields: Connected App Name (IntraSalesforceApp), API Name (IntraSalesforceApp), Contact Email (dev@valence.app), Contact Phone, Logo Image URL (with a link to 'Upload logo image or Choose one of our sample logos'), Icon URL (with a link to 'Choose one of our sample logos'), Info URL, and Description. The 'API (Enable OAuth Settings)' section includes: 'Enable OAuth Settings' (checked), 'Enable for Device Flow' (unchecked), 'Callback URL' (https://example.com), 'Use digital signatures' (unchecked), 'Selected OAuth Scopes' (two scopes: 'Access and manage your data (api)' and 'Perform requests on your behalf at any time (refresh\_token, offline\_access)'), 'Require Secret for Web Server Flow' (checked), 'Introspect All Tokens' (unchecked), 'Configure ID Token' (unchecked), 'Enable Asset Tokens' (unchecked), and 'Enable Single Logout' (unchecked).

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

[Edit](#) [Delete](#) [Manage](#)



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	3MVG9Yb5IqknkB4oS90nrVZ5im3yH27yQlqiWgIGMKKHJMByJew94s81sdSpo12ImAsFRciBUPicRukZ6dOU	Consumer Secret	<a href="#">Click to reveal</a>
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	<input type="checkbox"/>	Require Secret for Web Server Flow	<input checked="" type="checkbox"/>
Introspect All Tokens	<input type="checkbox"/>	Token Valid for	0 Hour(s)
Include Custom Attributes	<input type="checkbox"/>	Include Custom Permissions	<input type="checkbox"/>
Enable Single Logout	Single Logout disabled		

## 2.9.2 Set Up The Auth Provider

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

### Auth. Provider

**Auth. Provider Edit** [Save](#) [Save & New](#) [Cancel](#)

Provider Type	Salesforce
Name	IntraSalesforce
URL Suffix	IntraSalesforce
Consumer Key	3MVG9Yb5IqknkB4oS9 <a href="#">i</a>
Consumer Secret	24921698567A5AB7C6 <a href="#">i</a>
Authorize Endpoint URL	https://login.salesforce.com/services/oauth2/authorize <a href="#">i</a>
Token Endpoint URL	https://login.salesforce.com/services/oauth2/token <a href="#">i</a>
Default Scopes	api refresh_token <a href="#">i</a>
Include Consumer Secret in API Responses	<input checked="" type="checkbox"/> <a href="#">i</a>
Custom Error URL	<input type="text"/>
Custom Logout URL	<input type="text"/> <a href="#">i</a>
Registration Handler	<input type="text"/> <a href="#">i</a>
	<a href="#">Automatically create a registration handler template</a>
Execute Registration As	<input type="text"/> <a href="#">i</a>
Portal	--None--
Icon URL	<input type="text"/>
	<a href="#">Choose one of our sample icons</a>

[Save](#) [Save & New](#) [Cancel](#)

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		Edit	Delete	Clone
Auth. Provider ID	0SO0x000004Dr6			
Provider Type	Salesforce			
Name	IntraSalesforce			
URL Suffix	IntraSalesforce			
Consumer Key	3MVG9Yb5lqqnkB4oS90nrVZ5im3yH27yQlqiwGfGMKKHMJMbyJew94s81sdSpo12lmAsFRciBUPicRukZ6dOU			
Consumer Secret	<a href="#">Click to reveal</a>			
Authorize Endpoint URL	<a href="https://login.salesforce.com/services/oauth2/authorize">https://login.salesforce.com/services/oauth2/authorize</a>			
Token Endpoint URL	<a href="https://login.salesforce.com/services/oauth2/token">https://login.salesforce.com/services/oauth2/token</a>			
Default Scopes	api refresh_token			
Include Consumer Secret in API Responses	<input checked="" type="checkbox"/> <a href="#">i</a>			
Custom Error URL				
Custom Logout URL				
Registration Handler				
Execute Registration As				
Portal				
Icon URL				

---


Salesforce Configuration		Edit	Delete	Clone
Test-Only Initialization URL	<a href="https://ruby-flow-2519-dev-ed.cs95.my.salesforce.com/services/auth/test/IntraSalesforce">https://ruby-flow-2519-dev-ed.cs95.my.salesforce.com/services/auth/test/IntraSalesforce</a>			
Existing User Linking URL	<a href="https://ruby-flow-2519-dev-ed.cs95.my.salesforce.com/services/auth/link/IntraSalesforce">https://ruby-flow-2519-dev-ed.cs95.my.salesforce.com/services/auth/link/IntraSalesforce</a>			
OAuth-Only Initialization URL	<a href="https://ruby-flow-2519-dev-ed.cs95.my.salesforce.com/services/auth/oauth/IntraSalesforce">https://ruby-flow-2519-dev-ed.cs95.my.salesforce.com/services/auth/oauth/IntraSalesforce</a>			
Callback URL	<a href="https://ruby-flow-2519-dev-ed.cs95.my.salesforce.com/services/authcallback/IntraSalesforce">https://ruby-flow-2519-dev-ed.cs95.my.salesforce.com/services/authcallback/IntraSalesforce</a>			
Single Logout URL	<a href="https://ruby-flow-2519-dev-ed.cs95.my.salesforce.com/services/auth/rp/oidc/logout">https://ruby-flow-2519-dev-ed.cs95.my.salesforce.com/services/auth/rp/oidc/logout</a>			

### 2.9.3 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** [Help for this](#)

« Back to List: Custom Apps



Edit
Delete
Manage

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	3MVG9Yb5lqgnk84oS90nrVZ5lm3yH27yQlqiwGfGMKkHMJbyJew94s81sdSpo12mAsFRciBUPicRukZ6dOU	Consumer Secret	<a href="#">Click to reveal</a>
Selected OAuth Scopes	Access and manage your data (api) Perform requests on your behalf at any time (refresh_token, offline_access)	Callback URL	<a href="https://ruby-flow-2519-dev-ed.cs95.my.salesforce.com/services/authcallback/IntraSalesforce">https://ruby-flow-2519-dev-ed.cs95.my.salesforce.com/services/authcallback/IntraSalesforce</a>
Enable for Device Flow	<input type="checkbox"/>	Require Secret for Web Server Flow	<input checked="" type="checkbox"/>
Introspect All Tokens	<input type="checkbox"/>	Token Valid for	0 Hour(s)
Include Custom Attributes	<input type="checkbox"/>	Include Custom Permissions	<input type="checkbox"/>
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.

## 2.9.4 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		Edit	Delete	Clone
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	<input checked="" type="checkbox"/> <a href="#">i</a>			
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			
		Edit	Delete	Clone

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

## 2.9.5 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.

Label

Name

URL

---

**▼ Authentication**

Certificate

Identity Type

Authentication Protocol

Authentication Provider

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

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

## 2.9.6 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

## 2.9.7 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
Ex-ample 1	Account	Org A	Local	No	Account	Org B	Remote	Yes (Org B)
Ex-ample 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!

## 2.10 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.

## 2.10.1 Definition

```
/**
 * Exception related to, and thrown by, Adapters.
 */
global class AdapterException extends Exception {}
```

## 2.10.2 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);
}
```

## 2.11 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*.

This class is only used by Adapters that implement the *SourceAdapterForPull* interface.

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.

- *Immediate*
- *Scopes*
- *Delay*
- *Locator*
- *No Records*

**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.

---

### 2.11.1 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;  
}
```

### 2.11.2 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 leverages **Batch Apex** to break 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.

### 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;
}
```

(continues on next page)

```
public MyScope(String requestId, Integer offset) {
    this.requestId = requestId;
    this.offset = offset;
}
}
```

### 2.11.3 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 (handy if you know for sure that it'll be a few minutes before things are ready).

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 5-10 minutes
    String fileId = generateFile();

    // ask Valence to call you back in 15 minutes
    return valence.FetchStrategy.delay(15, 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);

    if(ready) {
```

(continues on next page)

(continued from previous page)

```

        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);`

## 2.11.4 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 [SourceAdapterForObjectPush](#) 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
}

```

### 2.11.5 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;  
}
```

## 2.12 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`.

### 2.12.1 Simple Field Example

```
valence.Field firstNameField = valence.Field.create('firstName').withLabel('First Name  
↳').build();
```

## 2.12.2 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();
```

## 2.12.3 Properties

Data Type	Class Property	Builder Method	Description
String	name		<i>Required.</i> Expected to match actual record field names.
String	label	withLabel	User-friendly label for this field.
String	description	withDescription	Description of the field's purpose.
String	exampleValue	withExampleValue	An example value to help give a user context.
String	defaultValue	withDefaultValue	The default value that will be used if this field is left blank.
String	type	withType	The data type of the field.
String	format	withFormat	The format of the value, if applicable.
Boolean	isCreateable	setCreateable	True if this field can be set on record creation.
Boolean	isUpdateable	setUpdateable	True if this field can be set on record update.
Boolean	isEditable	setEditable	True if either isCreateable or isUpdateable have been set. Can also be set independently.

## 2.13 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.

### 2.13.1 Definition

```
/**
 * Exception related to, and thrown by, Filters.
 */
global class FilterException extends Exception {}
```

### 2.13.2 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);
}
```

## 2.14 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.

## 2.14.1 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	linkSource-Name	The API table name of the source Table, if one has been defined.
String	linkSource-Label	The user-friendly label of the source Table, if one has been defined.
String	linkTarget-Name	The API table name of the target Table, if one has been defined.
String	linkTarget-Label	The user-friendly label of the target Table, if one has been defined.
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	sourceAdapter-Namespace	The namespace of the Adapter that is being used as the source Adapter.
String	sourceAdapter-ClassName	The class name of the Adapter that is being used as the source Adapter.
String	targetAdapter-Namespace	The namespace of the Adapter that is being used as the target Adapter.
String	targetAdapter-ClassName	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	batchSize-Limit	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.
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 <b>last-SuccessfulSync</b> and before <b>now</b> .
Map<String, Mapping>	mappings	All active mappings for this Link. <i>Inactive mappings will not be in this collection.</i>

## 2.15 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*.

## 2.15.1 Properties

Field	Description
name	A unique name for this mapping, comes from the DeveloperName field of the corresponding cMDT record.
sourceField-Name	The API name of the field on the data source.
targetField-Name	The API name of the field on the data target.
configuration	Configuration information, if any has been specified for this mapping in this context.

## 2.16 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.

### 2.16.1 Record Data

We think of each data record as a simple map of key-value pairs. We represent this in Apex using a `Map<String, Object>`. RecordInFlight internally uses two of these maps, one with the original properties that it was first created with, one with the properties as they are manipulated and changed by *Filters*.

#### RecordInFlight Data Methods

```
// constructor
global RecordInFlight (Map<String, Object> originalProperties);

global Map<String, Object> getProperties();

global Map<String, Object> getOriginalProperties();
```

#### Working with Record Data

```
Map<String, Object> props = new Map<String, Object>{
    'firstName' => 'Tom',
    'lastName' => 'Sinatra'
};

// creating a RecordInFlight
valence.RecordInFlight tom = new valence.RecordInFlight (props);

// accessing the original properties
System.assertEquals('Sinatra', tom.getOriginalProperties().get('lastName'));

// accessing properties that have possibly been changed
tom.getProperties();
```



## 2.16.2 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);
```

## 2.16.3 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);
```

## 2.16.4 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();
```

## 2.16.5 Salesforce Id

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();
```

## 2.17 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.

### 2.17.1 Simple Table Example

```
valence.Table companiesTable = valence.Table.create('company').withLabel('Companies').
    ↪build();
```

### 2.17.2 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.
Boolean	isCreateable	setCreateable	True if new records can be created in this table.
Boolean	isUpdateable	setUpdateable	True if existing records can be updated in this table.
Boolean	isEditable	setEditable	True if either isCreateable or isUpdateable have been set. Can also be set independently.

## 2.18 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()`

## 2.18.1 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();
}
```

## 2.19 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 *Using Dynamic Configurations in your Extension*.

---



---

**Note:** For example usage, see how we allow users to change their upsert field with our example in *ConfigurableTargetAdapter*.

---

### 2.19.1 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
     *
     */
}
```

(continues on next page)

(continued from previous page)

```

    * @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 configuration The raw configuration
     *
     * @return A human-readable and friendly explanation that specifically reflects
     ↪and explains the configuration passed.
     */
    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);
}

```

## 2.20 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 *Using Dynamic Configurations in your Extension*.

---

## 2.20.1 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
     *
     * @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 configuration The raw configuration
     *
     * @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.
     *
     */

```

(continues on next page)

(continued from previous page)

```

    * @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);
}

```

## 2.20.2 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>';
        }
    }
}

```

(continues on next page)

```

    }
}

public void setTargetConfiguration(valence.LinkContext context, String
↪configurationData) {
    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);
    }
}
}

```

## 2.21 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().

### 2.21.1 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
     */
    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.

## 2.22 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 **RequiresNamedCredentialForSchema\_\_c** is unchecked, `setNamedCredential()` will not be called before `getTables()`. Basically, `setNamedCredential()` is contextually-sensitive and will only be called when it is needed.

## 2.22.1 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.
     *
     * @see https://developer.salesforce.com/docs/atlas.en-us.apexcode.meta/apexcode/apex\_callouts\_named\_credentials.htm
     *
     * @param namedCredentialsName The API name of a NamedCredential defined in this
     * Salesforce org
     */
    void setNamedCredential(String namedCredentialName);
}
```

## 2.22.2 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
}
```

## 2.23 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*.

**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.

### 2.23.1 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
     * consideration.
     */
    List<Field> getFields(String tableApiName);
}
```

## 2.23.2 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()
    };
}
```

## 2.23.3 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;
}
```

## 2.24 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*.

### 2.24.1 Definition

```
/**
 * Implement this interface if your Adapter will be used in a Pull Model as a source_
↳ of records. The most common
 * example of this is an Adapter that calls an external system 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
```

(continues on next page)

(continued from previous page)

```

    * 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
    */
    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);
}

```

## 2.24.2 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.

To decide what you should be querying 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.mappings** - this is a List of `mappings` that hold the fields that should be queried from the source table. Records are given to Valence as `RecordInFlight` instances. The constructor for this DTO class expects a `Map<String, Object>` where the keys are the field names from the source system.

---

**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](https://developer.salesforce.com/docs/atlas.en-us.apexcode.meta/apexcode/apex_gov_limits.htm)

---

### 2.25 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.

### 2.25.1 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 part of a Push Model.
 */
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);
}
```

## 2.26 SourceAdapterForObjectPush

This interface allows your *Adapter* to be the start of a Push Link that is processing records from Salesforce.

Valence's LocalSalesforceAdapter implements this interface, so it's unlikely you'll ever have to implement it yourself but it's here if you need it.

### 2.26.1 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 part of a Push
 * Model.
 */
global interface SourceAdapterForObjectPush extends SourceAdapter {

    /**
     * Given a collection of sObject records, produce a collection of RecordInFlight
     * records that are ready for further
     * processing by the Valence engine.
     */
}
```

(continues on next page)

(continued from previous page)

```

*
* @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);
}

```

## 2.26.2 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;
    }
}

```

## 2.27 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.

---

### 2.27.1 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.
*/

```

(continues on next page)



(continued from previous page)

```

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
     *
     * @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
     *
     * @return A rehydrated instance of your scope class
     */
    Object deserializeScope(String serializedScope);

}

```

## 2.27.2 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;
}

```

## 2.28 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.

## 2.28.1 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.
     */
    void pushRecords(LinkContext context, List<RecordInFlight> records);
}

```

## 2.29 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.

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 *Using Dynamic Configurations in your Extension*.

---

### 2.29.1 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.

```

(continues on next page)

(continued from previous page)

```

*/
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.
     *
     * @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);
}

```

## 2.29.2 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)) {

```

(continues on next page)

(continued from previous page)

```

        // this mapping has a configuration for your filter
    }
}
}

```

## 2.30 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*.

### 2.30.1 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 avoid processing a record by
     * calling ignore() on it.
     * 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
     */
}

```

(continues on next page)

(continued from previous page)

```
void process(LinkContext context, List<RecordInFlight> records);  
}
```

### 2.30.2 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;  
    }  
  
    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');  
    }  
}
```