The Sandpiper Framework Reference Documentation

Introduction

About Sandpiper

Sandpiper establishes a common, decentralized method to classify, distribute, and synchronize product data between a canonical sender and a derivative receiver. To do this it defines, as unambiguously as possible, both a model for interaction and shared vocabulary to describe the many moving pieces involved.

Sandpiper tries to do this one thing well, and does not attempt to branch into other realms better handled by dedicated tools.

Background

This is the reference documentation for v0.9 of the Sandpiper Framework, a cross-platform, open-source product data synchronization initiative by members of the automotive aftermarket. Increasingly, product data resides in more systems, is more difficult to update, is less verifiable, and requires increasingly variable and proprietary methods to deliver.

The founding members of the team come from the automotive aftermarket industry, where the broad range of products sold (from consumer electronics to pistons and everything between) combine with stringent certifications of fitment and detail to create massive catalogs of data that must be updated regularly. A medium-sized aftermarket supplier will have tens of thousands of SKUs, hundreds of thousands of pictures, and millions of rows of fitment data to communicate to dozens of receivers monthly. To complicate matters further, no unambiguous standard for partial data delivery exists, meaning all of this data has to be sent in full to propagate even a single change.

Yet while this project began in the automotive world, the problem is one that extends to all product data, regardless of industry; though various industry- and partner-specific standards and formats exist to describe products, there's no standard way to actually *send* them, to change just one piece of one product's data, or to make sure that what *was* sent actually covers what was requested. This applies as much to T-shirts as it does to spark plugs.

We believe the Sandpiper framework can make this process a little less painful for everyone who has to get information about their products into the world.

Basic Terms

Before we go further, there are a few basic terms to introduce, since they're used so often. For more detailed explanations, you can refer to the later parts of the document.

In Sandpiper, individuals or individual systems involved in exchanging and hosting data are known as *Nodes*. When they interact, these nodes are known as *Actors*.

Actors exchange data about *Products*. Products (or SKUs, units, items, parts, and so on) are usually goods -- though they can also be services. Sandpiper specializes in the core data that defines these products, which we call *Product Data*: information that, were it to change, would also mean the product or its use itself had changed.

The framework does not make special accommodations for other kinds of data, which we call *Non-Product Data*, even when it is product adjacent.

Data & Object Models

Products & Product Data

Products

Products are controlled by a *Creator*, the manufacturer or provider with ultimate authority over its form and availability. A product has a single *Part Number* that is unique among all its creator's products[1].

Product Data

Sandpiper's focus is product data, which has two primary characteristics:

1. It defines the product, such that changing existing product data (except as a correction or addition) usually means a material change to the product itself
2. It is nearly stateless, in that it changes infrequently, is not time-sensitive, and is not context-sensitive

To give an example of the first primary characteristic of product data, if a sprocket has five teeth when first communicated, that could not be changed to six teeth in a second communication without raising an eyebrow; if the part truly changed from five to six teeth, it's been fundamentally altered and will not function the same way. It really is a new product, even if it supersedes the old for some reason. While Sandpiper doesn't prohibit any changes, making functional modifications like this without introducing a new number is at best ill-advised.

To illustrate the second primary characteristic of product data, if the same sprocket is communicated at the same time to Customer A and Customer B, they will both see the same number of teeth. Product data is the same at any given point in time for any customer or relationship[2].

Some examples of product data:

- Fitment information
- Physical characteristics like weight, dimensions, etc.
- Pictures of the product that convey its characteristics
- Product contents list
- Permanent marketing copy (i.e. not campaign copy and the like)
- Universal retail price

Non-Product Data

Sandpiper doesn't forbid the transmission of other types of data (particularly since there's so much grey area), but doesn't make any provisions for it.

In contrast to product data, other kinds of data tend to change quickly, or are only valid within a given time or context; they are stateful and reference rather than define products. This includes the data that records and enables purchasing between entities as well as that which describes the current status of products within a supply chain.

Some examples of non-product data:

- Per-customer pricing
- Market campaign copy
- Purchase orders
- Inventory reports

Grey Areas

In some scenarios, product data begins to approach non-product data, particularly when dealing with non-critical attributes. For example, for purely functional products like oil seals, the color is most likely not critical to the part's function, and sometimes this changes frequently. To remain flexible in these cases, Sandpiper doesn't enforce true statelessness of the product itself, only of the data at any given point in time.

As far as Sandpiper is concerned, a change to any product data creates a new state or revision of that product's data, rather than creating a new product. The creator and part number are the only elements that can't change without the product being considered new.

Data Synchronization

Sandpiper achieves its goal of reproducible, atomic data synchronization by strictly defining updates as either full replacements of all known data within a well-described set, or as additions and deletions of individual records within those sets using universally unique IDs (UUIDs).

Full Replacements

One way to assure reproducibility is by simply replacing all of one type of data in one go, for example, all fitment data for water pumps. The sender provides a complete universe, and the receiver is expected to more or less remove their old data and replace it with the new.

This does not provide a reliable way to update data in smaller pieces, and the scale of these updates becomes so large that it's not practical to do so more frequently than once a day at the most. For very large sets this can even be quarterly or yearly.

Full replacements also need to specify and match their scopes carefully; if the receiver's understanding of what they should delete is broader than what they receive, they'll drop data that will not be replaced.

Sandpiper's full replacement model can only be used in Level 1.

Partial Changes

Partial changes in theory allow for high-frequency updates, pinpoint corrections, and easy expansions of data. Existing ways to do this, however, fail to meet Sandpiper's synchronization goal in two major areas: fragility of methods and uncertainty of state across systems. To address these two areas Sandpiper uses UUIDs representing unchanging data records that can only be deleted or added.

Within a set, the individual records within a pool are immutable, i.e. once defined, they cannot be changed. Thus a unique ID will always refer to both the same values and the actual data record containing them. In this way sets of data can be mathematically compared and resolved, and the end state of a second dataset will always provably match the state of the first.

Sandpiper's partial change model can only be used in Level 2 and higher.

Why not "Delete/Update/Add"?

Adding information to an existing dataset is well understood; the new data augments the old, and no modifications to existing data are required.

Removing information is a little more complicated because to do so the remover needs to specify exactly what existing data needs to be deleted, as well as what to do when there are multiple records sharing that same specification. Whole chains of validation exist today to attempt to resolve these deletes based on values, and missing values or things like different character encodings create huge headaches.

Updating information is even more complicated. On top of all the same requirements that would be present for a delete, replacement values must also be provided, and then subjected to the same validation as adding new information. Without doing this, values within a record could be updated to create overlaps and violate domain rules. After the fact, there's also no way to know if a record has been changed without comparing all of its values to the previous record -- which may have changed itself. So clearly when processing updates as field-level changes in product data, there's no way to know the exact state of any dataset without examining *all* of the dataset.

Therefore a record update in a dataset is actually at least as difficult to orchestrate as a delete *and* an add, but without any of the certainty.

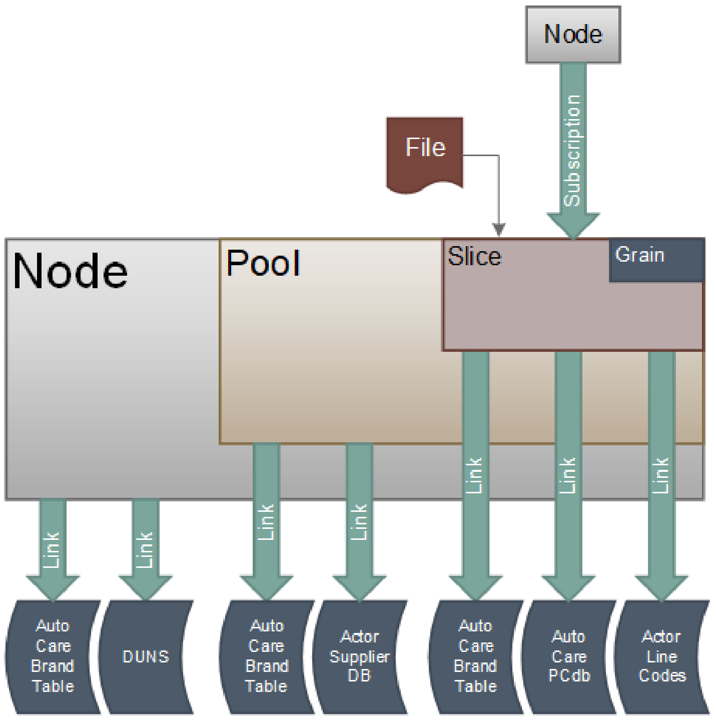
For these reasons Sandpiper avoids updates except as a concept, and actually treats changes as deletions and additions. UUIDs within a set cannot be reused even when a record is identical in form, however, because the ID itself also represents the time and metadata of the original.

Object Model

The object model for Sandpiper defines a set of common abstractions for the product data each node stores. There are just four persistent objects (*Node*, *Pool*, *Slice*, and *Grain*) and two reference objects (*Link* and *Subscription*). All Sandpiper objects have a universally unique ID that will be used for actions exclusively whenever possible; in the attribute lists in each section here, this has been omitted, but is assumed.

The node represents the root of one self-contained Sandpiper environment, with one controller. It contains pools of product data, each with one owner. These pools are further subdivided into slices, each representing one set of the same type of data and specifying how it is internally organized. That data is finally broken into grains by the method of organization named on the slice.

To structure this data and aid the creation of shared scope between actors, the persistent object types can employ links: references to additional systems, descriptions, and data. To create the bond between actors, the secondary actor establishes subscriptions to the slices available.



Persistent Objects

Nodes

The node is a single Sandpiper instance or system[3]. It has a *Controller* responsible for, though not necessarily the originator of, its operation and contents.

Note: a human interacting at Level 1-1 is technically a node, though their data state is unknown after retrieval.

Attributes

Name	Plan Attribute	API Attribute	Presence	Allowed Values	Example

Pools

Within each Sandpiper node, product data is stored in broad banks called *Pools*. These represent a business or management-level division, so that a single node might contain product data spanning multiple business approaches yet being coordinated within one system.

While a node has a controller, a pool has a *Creator*, the owner of the product data within. In some cases this will be the same as the controller, and in others it will be different. For example, if the node operator works for a shared services provider that offers data synchronization for multiple customers, the controller will be the provider, and the creator will be the customer.

Pools can be one of two types: *Canonical* or *Snapshot*.

A node's canonical pools contain the data that it owns and controls; changes made to a local node's canonical pools can be transferred to external Sandpiper nodes, with the local node as the origin point.

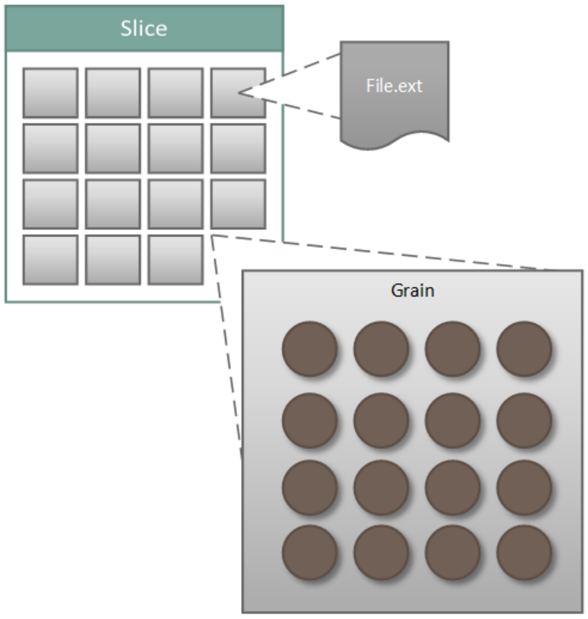
A node's snapshot pools contain copies of the data in other nodes' canonical pools transferred in this way. A snapshot pool is just that: a snapshot of some or all of the data in a canonical pool from an external node, at its last-known state.

Attributes

Name	Plan Attribute	API Attribute	Presence	Allowed Values	Example

Slices

A pool is divided into *Slices*. The slice is the fundamental unit of Sandpiper; basic transactions are expected to operate only on the slice, and it provides the context for all more complex transactions as well. In some cases it can be thought of as the file level of the data.



A slice defines the single type of the data it contains (see the slice types list). All grains within a slice must be the same type. The slice also defines a filename for Level 1 transactions.

Attributes

Attribute	Plan Attribute	Presence	Allowed Values	Example
Unique ID	uuid	required	UUID	a4380ad7-31a7-4f31-864d-4c8de746ad6f
Slice Data Type	slicetype	required	Sandpiper slice types list	aces-file
File Name	filename	optional	alpha string	ApplicationFile.xml

Grains

A slice is broken into *Grains*, each representing one unit of meaning or scope. It can be thought of as the element level of the data.

Grains have a *Grain Key* containing a single text value, to safely and atomically operate on the data in pieces smaller than a whole slice. This value must be a single unicode key that directly references one key value within the data, e.g. a part number or a UUID. It must not be an artificially packed or delimited set of values referring to more than one key within the data.

The grain is the smallest unit on which a Sandpiper node can act, and can only be directly addressed in Level 2 and higher transactions.

Grains are not part of the plan; they reside below the slice, the lowest level of agreement between actors.

Attributes

Name	Plan Attribute	API Attribute	Presence	Allowed Values	Example
Grain Key	*none*	grain_key	required	unicode string	"23490A"

Reference Objects

Links

Links are references that allow slices to be tied to other systems and tagged with nonstandard metadata.

The link is the primary means of attaching overarching structure to slice data. Every partnership will have a different preferred method for establishing things like line codes, hierarchies, and sets, so the link provides a few standard methods to do this and an extensible category for what it doesn't define.

The link is also the way Sandpiper connects slice data to description or validation frameworks like reference database versions, business identities, and so on.

Attributes

Name	Plan Attribute	API Attribute	Presence	Allowed Values	Example
Key field reference pointer	keyfield	?	required	Sandpiper Key Field List	"autocare-pcdb-partterminology"
Key field value	keyvalue	?	required	Text	"9999922"
Description	description	?	optional	Text	"Yak Milk Filter"

The Root-Derivative Slice Group

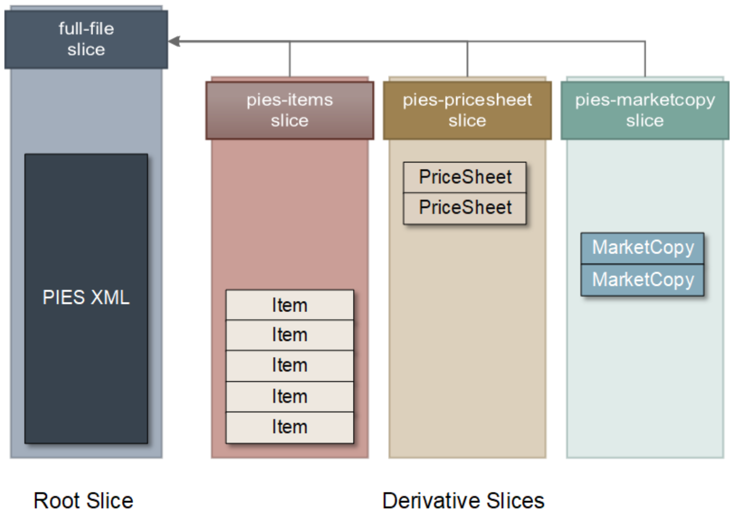
A Level 1 slice contains a full file that can only be communicated in full. But what if we want to break it into parts that can be synchronized individually? This becomes possible in Level 2 transactions, where we can more finely subdivide the data in a full file and use these pieces to deterministically synchronize primary and secondary data pools.

To do this cleanly, we'll define a root slice and one or more derivative slices, using links to designate the relationship. This is know as a *Root-Derivative Slice Group*, and it allows a Sandpiper node to support both Level 1 full file slices and Level 2+ operations on that same data.

All slices technically contain at least one grain; in the case of a Level 1 file, it's just that there's one single, large grain: the full file. But for Level 2 slices, the information in the full file can be broken into grains that allow different perspectives of access and synchronization, e.g. by PIES item, by ACES item, and more.

In the root-derivative group pattern, we define a single Level 1 *Root Slice* with the full dataset, and one or more *Derivative Slices* containing granulated data extracted from the master. The derivative slices reference the root's unique ID using a link with system "root", indicating their source. These derivative slices use the *Slice Type* to indicate the kind of division they do (i.e. what kinds of grains are within), and actors can subscribe to them directly.

For example, in a PIES file, we might want to work with PIES items (which will use the part number as a key) as well as PIES market copy and PIES price sheets (which do not reference part numbers and have to be keyed differently). In this case we'd have four slices: the full root, a derivative slice by items, a derivative slice by price sheets, and a derivative slice by market copy.



In this way, a file with multiple segments can be broken out for processing at Level 2, yet Level 1 remains accessible and useful for full set analysis, for file areas that cannot be easily granulated, like prerolls and headers, and for use in integration environments where the full file must be used for some branches even though granular updates can be used for others.

Further, this pattern allows external granulators (triggered either manually or by a detected change in the database) to automatically update a derivative slice using the updated root information. This reduces process dependencies and ensures reliable change propagation in a batch environment.

Subscriptions

The secondary actor in a Sandpiper relationship can subscribe to a slice, stating its intention to mirror that data and keep it synchronized with the primary actor.

This subscription includes the secondary actor's stated preference for receipt of the data, particularly the frequency of synchronization, aka the period. Actors should not attempt to synchronize more frequently than the minimum period, and should define their own preferences around what load their infrastructure can safely handle. In future versions, this may also include whether it should be pushed or pulled, what methods should be employed, what schedule should be followed, and what credentials will be used.

Attributes

Name	Plan Attribute	API Attribute	Presence	Allowed Values	Example
Slice UUID	sliceuuid	?	required	uuid	cebafea0-421b-4dba-8b9e-d9cb60ce41de
Minimum Period	minperiod	?	required	decimal	1.5
Minimum Period UOM	minperioduom	?	required	"seconds","hours", or "days"	days

Granulation

The Sandpiper server is focused on delivery and receipt of data to preserve one source for the unambiguous truth around what is available for use. It is both a repository for the product data and a mechanism for resolving differences between multiple repositories.

Because this is its focus, the server does not attempt to parse or understand the data it contains -- only its state. This means that the server itself will not be able to perform the rendering process to create grains.[4]

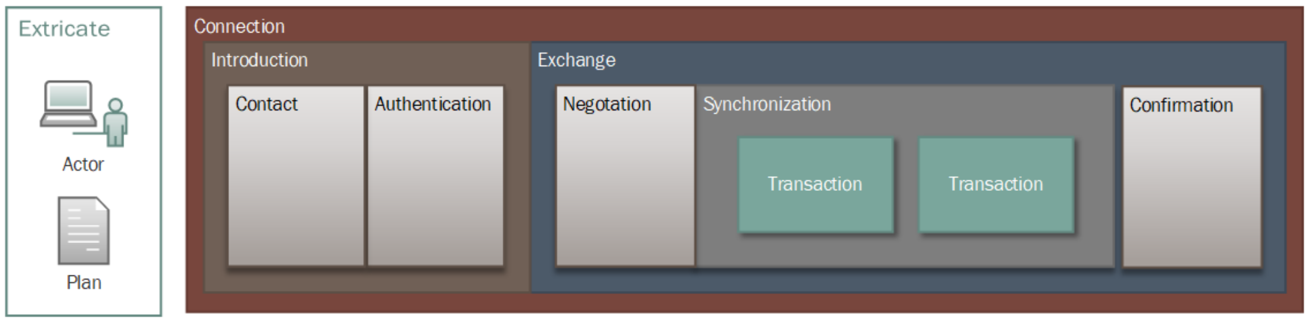
For Level 1, which operates at the slice only, this makes no difference; the grain is never engaged. For Level 2 and higher, though, Sandpiper exposes commands to input grains, allowing external toolchains (for short, called *Granulators*) to parse full data into grains.

In these sections, we'll refer to the source of the original data as the *PIM:* the Product Information Manager.[5] The Sandpiper framework classifies PIMs in three tiers:

Tier	Description
Classic	Output of single files that are communicated traditionally via email, FTP, or web portal
Sandpiper-Aware	Can execute Sandpiper commands and supporting tools externally, though not query the data or use the API directly
Sandpiper-Capable	Can transfer information directly into a Sandpiper server and query the data to make intelligent decisions about updates
Sandpiper-Native	Includes an embedded Sandpiper instance that directly generates UUIDs and tracks changes, granulates as desired, etc.

Interaction Model

Sandpiper's main goal is to facilitate repeatable, deterministic data transfer, and to do this it lays out a model for node interaction.



1. Outside well-defined communication channels, objects and roles are established. This segment is known as the *Extricate.*
    1. A system or human employing Sandpiper is known as an *Actor*.
    2. Before engaging, two actors must establish a governing agreement known as the *Plan*.
2. All actor communication via Sandpiper occurs within a *Connection*.
    1. Prior to beginning substantive data transfer, actors begin with the *introduction*.
        1. Actors make *Contact* via the network
        2. Actors *Authenticate* to verify their respective identities
    2. Once the introduction is complete, actors transfer product data during the *Exchange*.
        1. Exchanges are established and next steps are unlocked through *Negotiation*.

2. Transferring product data and resolving pools as part of an exchange is known as *Synchronization*.
    1. Two Actors' operations and communications during synchronization are part of one or more *Transactions*.
3. After synchronization, actors communicate about the data exchanged and sign off on the results during *Confirmation*.

Data transfer is only one-way: in any connection only one actor will receive product data.

Actors

In the context of a connection, nodes, humans, and systems assume a role as an actor. Any connection has only two actors: a *Primary Actor* and a *Secondary Actor.*

The primary actor is the sender of data, responsible for providing information about and issuing updates from its canonical pools.

The secondary actor is the recipient of data. This actor can be a human or a full Sandpiper node. The former is known as *Basic Secondary Actors*, because it cannot engage in a true Sandpiper exchange, and the latter are known as *Advanced Secondary Actors*. Advanced secondary actors are responsible for providing information about their snapshot pools as well as processing updates provided by the primary actor.

The Plan

The agreement between two actors is known as the Plan. This describes and confirms a shared understanding of the process, structure, and metadata surrounding synchronization, though not the product data itself. The Sandpiper API and management interfaces actually implement these details -- the Plan itself is not a method for establishing agreements, modifying slice scopes, and so on. Instead, the Plan represents the state of these facets at a given point in time, and the *Plan Document* encapsulates them in an implementation-independent XML format that can be compared between actors and signed off on by both.

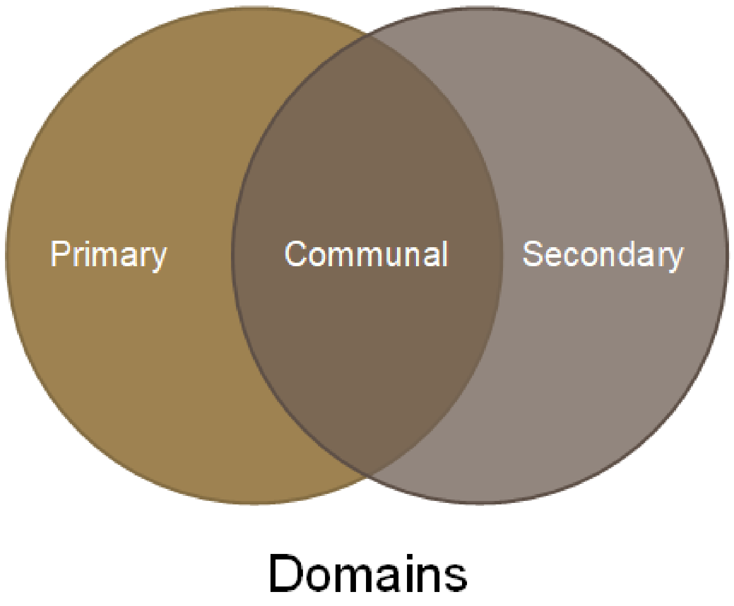
The plan itself has a UUID that uniquely identifies it among all plans in use globally.

Either actor may terminate the agreement. Failure to agree on a proposed change also results in a failed plan, and it must be put on hold until resolved.

Plan Domains

A Sandpiper plan establishes three domains and their ownership: The *Primary Domain* (owned by the primary actor), the *Secondary Domain* (owned by the secondary actor), and the *Communal

Domain* (jointly shared between primary and secondary). To modify any part of the plan, both actors must agree to the change.



Primary Plan Domain

The primary domain describes all the objects required in a partnership that reside with the primary node. This includes the node itself, its capabilities, its controller, the data pools, etc., and all the links to other systems that they have. Only the primary actor may modify this information.

Secondary Plan Domain

The secondary domain mainly describes the secondary actor, providing node information and links to other systems, capabilities, preferences, and so on. Only the secondary actor may modify this information.

Communal Plan Domain

The communal domain includes the subscriptions between the primary and secondary actors, and in the future may include other information to facilitate the relationship like cryptographic signatures. Either the primary or secondary actor may modify this information, but must propose this change through the Sandpiper API and wait for acceptance from the other party. If this acceptance is not given, the plan is placed on hold and no further synchronization can take place.

The Connection

The connection encloses all Sandpiper framework communication between actors. Note that it is predicated on information already established as part of the extricate.

The Introduction

In the introduction, one actor contacts the other and proves its identity so that the exchange can begin.

Contact

One actor, known as the *Initiator*, connects to the other, known as the *Respondent*. At this point, neither actor is technically primary or secondary -- that relationship will be established later, when the two actors decide the course of action they'll be taking. A given actor could receive data for one set of product information, or send data for another; this is dictated by the plan, which will be agreed to in the negotiation.

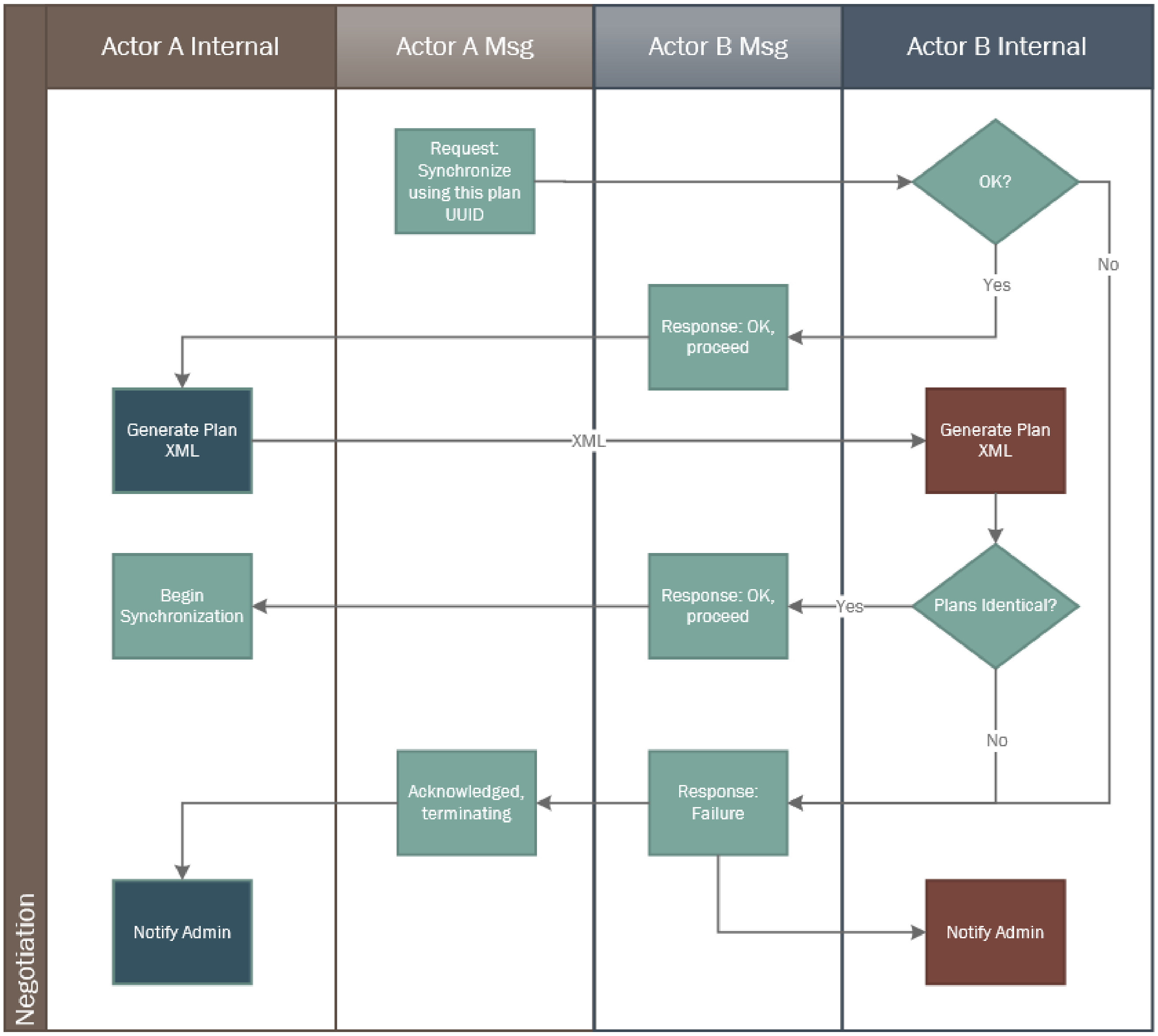
Authentication

In the authentication, the initiator supplies credentials to the respondent that prove its identity.

The Exchange

An exchange begins when two actors successfully complete the introduction. These actors enter into negotiation, where they verify that their understandings of the plan are the same before proceeding.

Negotiation



Negotiation attempts to establish an unambiguous course of action, or to safely discontinue the exchange so that administrators can update their systems. If the negotiation fails, the connection is aborted and both actors' administrators are notified of the discrepancy so that humans can resolve the issue.

In this step, the initiator proposes proceeding using a given plan, referencing its UUID. The respondent can agree to proceed with this plan; if it does not agree, the connection is aborted.

When the respondent does agree to proceed, the initiator generates a plan document that represents its current understanding of this particular plan, then transfers it to the respondent, who does the same, and compares the two documents. If they are not identical in content[6], the exchange can't continue, and the connection is aborted.

Transaction

After agreeing to the plan, the actors now assume the roles (primary or secondary) specified by the plan.

Using the subscription UUID, the secondary actor requests synchronization of data. This subscription specifies the method for that synchronization, and the actors execute it.

Confirmation

Once synchronization is complete, the actors confirm their satisfaction with the data state and disconnect.

Levels

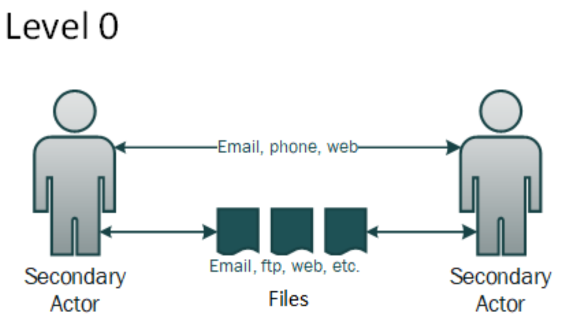
Sandpiper defines common minimum thresholds of capability for systems, so that simple needs can be met easily by basic implementations, and advanced needs can be met with more advanced implementations. In Sandpiper, these capabilities are grouped into *Levels*, with the lower levels having less functionality and the higher more.

Higher levels inherit the capabilities of the lower levels, and are aware of the elements defined in them. However, to maintain sanity, they cannot normally modify the data of lower levels; as the interaction proceeds, Sandpiper conceptually steps up and down the levels, so that a Level 1 interaction is the first initiated.

Some levels may have sub-levels; these will be written in the format L-n, where L is the level and n is the sub-level. For example, Level 1-1.

Level 0

Though not actually actionable within Sandpiper, the framework defines a prototypical Level 0 to represent uncontrolled product data exchange. Level 0 represents human-to-human interaction where actual files are sent between humans operating computers, who make agreements between themselves about how these files should be processed, their scope, and so on.



The only mechanism for this exchange is human-to-human.

Level 1

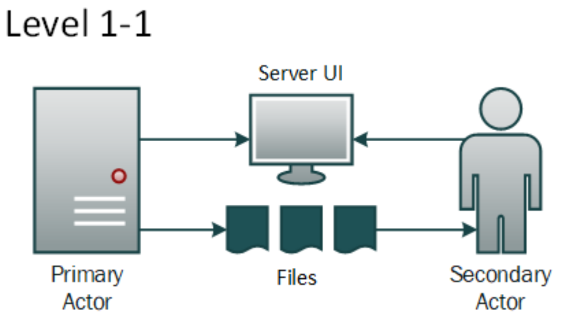
Level 1 is the first and simplest method of communicating product data. Information is exchanged in complete collections as files, which must replace all of the data stored at the recipient.

Level 1 is equivalent to sending full files between partners manually, but with the benefits of the Sandpiper framework's metadata, automation and validation.

This level is periodic and delivery-based. It has two sub-levels to serve either human-machine or machine-machine interaction; this must be chosen during negotiation.

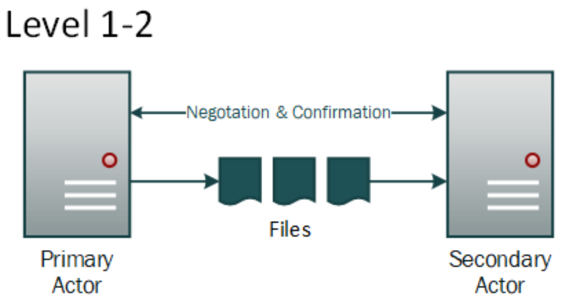
Level 1-1: Basic Exchange

A Level 1-1 basic exchange begins with the plan but never proceeds into synchronization; it allows a human to connect to a machine, complete the plan, and retrieve full files. Currently the only supported method for Level 1-1 is a human connecting to a Sandpiper server's web UI as a data portal.



Level 1-2: Advanced Exchange

Level 1-2 advanced exchanges can only occur between two Sandpiper nodes, machine-to-machine. They do not not use portal-driven delivery; rather, nodes transfer files directly between nodes using the Sandpiper transport.



Level 1 Negotiation

To start, in Level 1, nodes declare themselves, define their capabilities, and agree on their actions. The chief mechanism for this definition is the plan, as an XML file passed back and forth, with the actors filling in their proposed states and accepting or rejecting the changes.

All subscriptions occur at the slice, using its unique ID. It is not possible to retrieve data at any higher or lower position in the object model at Level 1; for that, Level 2 and beyond must be engaged.

A secondary actor cannot subscribe to or see snapshot pools held by the primary actor.

Level 1 Delivery

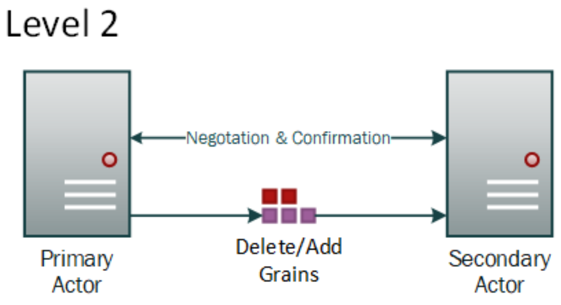
Through its subscriptions, the secondary actor indicates its preference for delivery of files containing all the data contained in one or more slices within the primary actor's canonical pools. It will receive or retrieve this data on a set schedule and via methods both defined in the plan.

Level 1 Integration

The secondary actor's node *must* archive or delete all previous data associated with the unique ID of the slices it received, replacing it in full with the new data received.

Level 2

Level 2 provides the ability to use an interface-driven mode where data may be modified in more targeted pieces, still in the periodic subscription paradigm but at a lower level.



The primary means of this interaction is via the Sandpiper API. This and further interactions must be performed machine-to-machine.

Level 3

Level 3 opens realtime communication of changes via a push mechanism.

Level 3 is not currently defined in detail; it will be part of Sandpiper 2.0.

The Sandpiper API and Protocol

At Level 1 and Level 2, the basic method of interaction in Sandpiper is through a RESTful HTTPS API employing Java Web Tokens (JWT). At Level 3, higher-order communication becomes possible and communication shifts to a WebSocket channel.

Sandpiper adopts this model for Level 1 & 2 both because it matches the periodic nature of these interactions and because the implementation of this pattern is well supported in most major programming environments. Level 3 introduces truly real-time bi-directional communication, with all its associated complexity; to require it for simpler modes would hinder adoption and create opportunities for errors.

Level 1 and 2 API

The respondent is the host for the interaction, and provides an API entry point URL that accepts the authentication request. All interactions must be completed over HTTPS.

In these levels, negotiation is combined with authentication because, in contrast to level 3, there is no need to establish authorization that persists across multiple plans as part of an overarching session.

The respondent has included, in the plan, a URI for accessing the server. This base URI is combined with the paths listed in each section to create an API location.

Authentication and Negotiation

Authentication is a simple username and password, submitted by the initiator to the respondent as JSON POSTed to the entry point URL, always the base URI plus "/login". This JSON object may also include a base64-encoded copy of the plan document representing that actor's understanding of the interaction's context.

The respondent must first verify that the username and password are valid, and then, if a plan document is present, compare this plan document to its own understanding of the plans available between the two actors. Messages (including errors) are returned inside the "message" attribute of a JSON string; successes will also include the JWT standard "token" and "expires" attributes.

The response may also include a planschemaerrors attribute to list any warnings or errors found with the plan schema.

Level 1 and 2 Authentication/Negotiation Request JSON Object

Attribute	Description
username	User Name
password	Password
plandocument	Base64-encoded Plan Document

Example:

```
{     "username": "bobauto"
    , "password": "password123"
    , "plandocument": "[... base64 content ...]"
}
```

Level 1 and 2 Authentication/Negotiation Response JSON Object

Attribute	Description
token	The JWT
expires	Expiration date and time
planschemaerrors	Issues or warnings around the plan schema
message	Response message or status text

Valid values for the message attribute:

Response	Description
Successful Authentication With Plan	Login successful
Authentication Error	Invalid username and/or password

The Level 1 and 2 Java Web Token

If authentication is successful, the respondent issues a token to the initiator that contains access rights information and paths for further interaction through the API. This JWT is subsequently delivered by the initiator to prove its identity as part of every request, through the "bearer" header.

By default this token should have a short expiration time -- 15 minutes or less -- and the initiator is responsible for requesting an extension when the token expires. It can do so by re-authenticating; since the state of any interaction is not tracked by the respondent, the initiator can just re-up before continuing to its next request.

As in any REST API, access should be checked when the request is made, not continuously during ongoing deliveries of information, like a long running download.

Level 3 API and Protocol

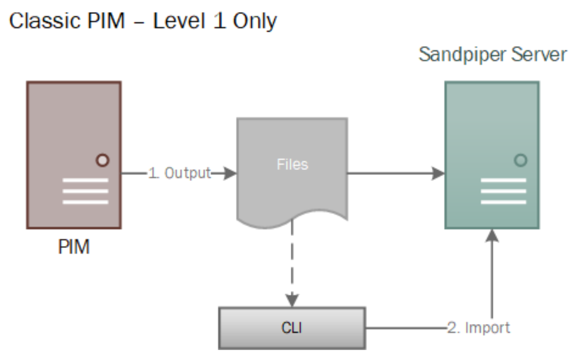
These will be implemented and expanded with Sandpiper 2.0.

Implementation Examples

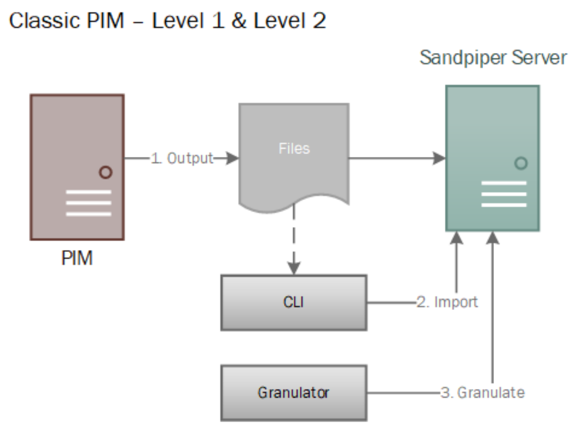
Input Workflow

The capabilities of the data source will direct the workflow that's most efficient. Classic PIMs require some user intervention, and more modern PIMs reduce that need.

Classic PIM Input

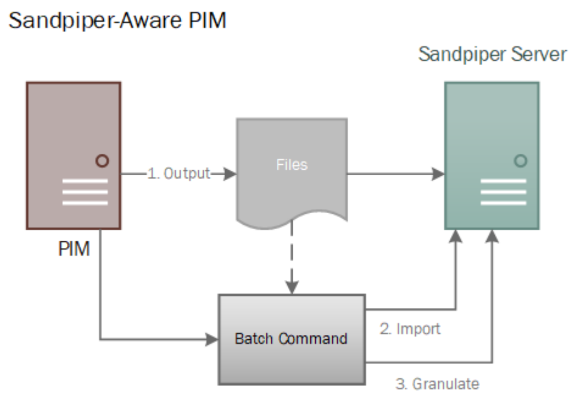


Level 1, being file-based, is designed for classic PIMs that can't use or haven't yet been adapted to use the Sandpiper framework. The PIM outputs files, and the user loads them into the Sandpiper server using the commandline interface.



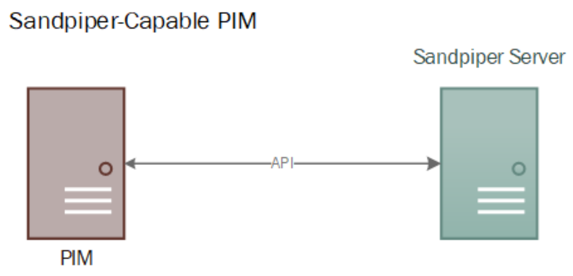
Level 2 introduces the ability to split complete datasets into grains. The server itself does not attempt to parse or interpret data, yet classic PIMs have no internal capacity to do this. Sandpiper is designed to support this scenario but to do so it will need an external, domain-specific tool to do so (called a *Granulator.*)

Sandpiper-Aware PIM Input



Sandpiper-Aware PIMs are able to use Sandpiper commands to do basic import and launch other tools. This may take the onus off of the user to manually import the data, though the process is likely only semi-automated.

Sandpiper-Capable PIM Input

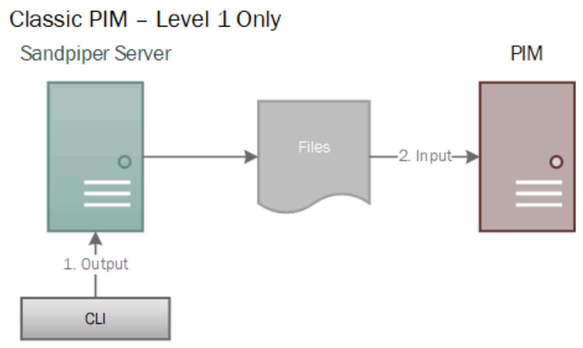


Sandpiper-Capable PIMs can communicate directly with the Sandpiper server, so for day-to-day operations the user does not need to engage any external tools while updating data.

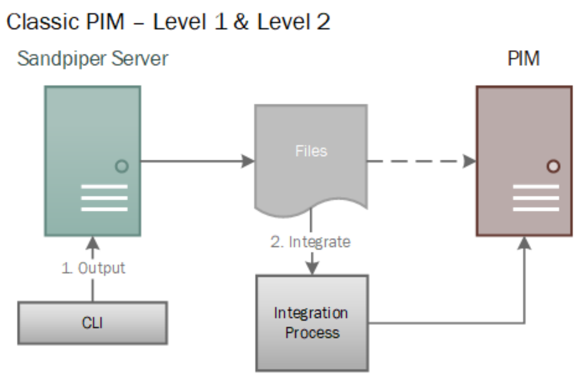
Output Workflow

As with input, the capabilities of the PIM receiver will direct the most efficient workflow.

Classic PIM Output

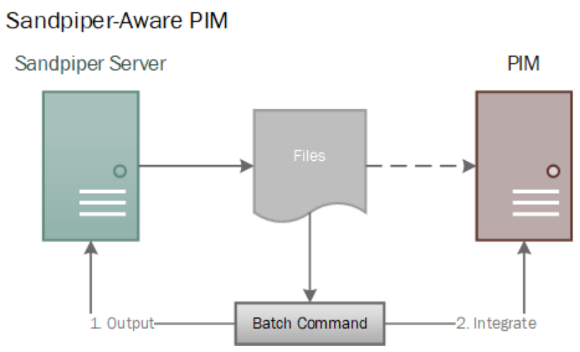


The classic PIM, without additional development, can make use of a purely Level 1 output process. The Sandpiper server, after synchronization with the primary Sandpiper node, outputs files via the CLI. The PIM then imports these using established processes.



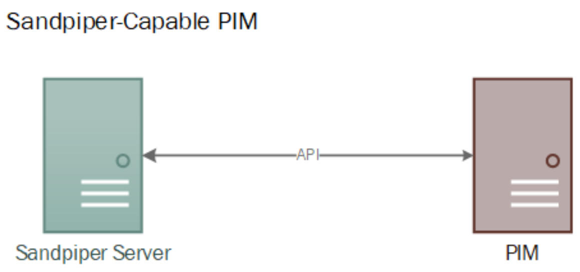
With an integration process, a classic PIM can also use the results of Level 2 transactions. More advanced recipients often already have a process to do something similar (for example, by comparing existing files to data in the PIM). Using the Sandpiper API and/or CLI, an external migration program can offload this change comparison to the deterministic Sandpiper framework, yet feed the PIM in the way it's already operating.

Sandpiper-Aware PIM Output



Sandpiper-Aware PIMs may not directly integrate Sandpiper into their logic, but can trigger regular loads and audits using external commands.

Sandpiper-Capable PIM Output



Sandpiper-Capable PIMs speak directly to the Sandpiper server via the API, integrating the functions so that no external tooling or user intervention is required.

Tips and Best Practices

UUIDs

IDs in Sandpiper are Universally Unique IDentifiers (UUIDs)[7].

Ensuring Uniqueness

A UUID should by its nature be unique across all nodes and all objects ever created; the odds are astronomically small that any overlap can occur (in more than 100 trillion UUIDs, the odds of any duplication are one in a billion).

However, unlikely does not mean *impossible*. Sandpiper implementations need to check these UUIDs when acting on data, and raise a fatal error if one is found. As each controller has a UUID, and each slice has a UUID, and the number of these will be small, a good first step is to check during the initialization phase against the node's already known IDs.

When receiving data, a similar check should be made against existing UUIDs tied to other data. Depending on the implementation's database schema, this would need to happen before automatically processing deletes, and constraints should be in place to ensure uniqueness of UUIDs so that overlapping additions can't be made.

Data Integrations

Pool Hygiene

Remember that the canonical pool is the internal data owned and controlled by a node, and the snapshot pools are the last-known state of other nodes' owned data. These are separate for a reason: the moment you integrate data, you change it in some way, even if all you're changing is the owner ID.

These snapshot pools are your system's way to make decisions about the state of external data outside your control. You can use that data to change your own, but that data is never the state of *your* data.

The canonical pools in a node must never directly accept data from another node. Resist the temptation to simplify maintenance by combining these pools behind the scenes.

Instead, the data received from another node should be integrated into your own data using a migration routine. This way you will identify many potential conflicts.

Staging Updates for Atomicity

Sometimes in the middle of an update systems will fail. At the least it's possible that a sudden hardware server fault could cause the machine to halt in the middle of a change.

Database servers have several ways to guarantee that these updates can be rolled back and re-applied. However, their methods may still leave a pool in an unknown state if the Sandpiper implementation doesn't properly use them.

Each Sandpiper transaction should be treated as an indivisible update; if one part of it fails, the whole thing should be rolled back. Properly boxing these updates can be done using SQL TRANSACTION statements, but as integration grows and crosses environments, it may not be enough. In these cases, instead of operating directly on the current tables, it makes sense to operate on copies of the data in staging tables. Once the transaction is complete, the tables can be swapped atomically. Another benefit of this is that indexes can be dropped and added on the staging table as needed for performance.

Appendix A: Reference Values

This section contains the valid reference values for this version of the Sandpiper specification.

Slice Type

Type	Level	Description	Grain Key References
aces-file	1	Auto Care ACES file	*none*
aces-apps	2+	ACES application elements	?
partspro-file	1	NAPA partspro file	*none*
napa-interchange-file	1	NAPA interchange file	*none*
pies-file	1	Auto Care PIES file	*none*
pies-items	2+	PIES item elements	Part Number
pies-pricesheets	2+	PIES pricesheet elements	?
asset-file	1	Single digital asset (image, video, etc.)	*none*
asset-archive	1	Archive of digital assets (image, video, etc.) in zip, gzip, etc.	*none*
asset-files	2+	Binary digital assets	Filename
binary-blob	1	Generic binary file	*none*

Type	Level	Description	Grain Key References
xml-file	1	Generic XML file	*none*
text-file	1	Generic text file	*none*

Glossary

Actor
A node, system, or human taking part in an exchange. Can be either primary or secondary

Advanced Secondary Actor
A Sandpiper node that can engage in a full exchange

Basic Secondary Actor
A human actor that can only retrieve or receive data as files

Canonical Pool
The data that an individual node owns

Confirmation
Interaction model step. In this step actors agree on the results of an exchange

Controller
The owner and operator of a node

Creator
The owner and/or originator of a product

Data Object
An individual item being acted on in a Level 2 transaction

Exchange
Interaction model. Information transfer between actors

Full Replacement
Scoped data that replaces all data previously held within the same scope. In Sandpiper this happens at the slice level

Grain
An addressable and self-contained unit within the slice

Grain Key

A reference to a single key value that can be used by Sandpiper to operate on data on a grain-by-grain basis

Granulator

External tool that parses data stored in full Sandpiper slices and extracts grains into associated granular slices

Initiator

A Sandpiper actor who initiates a connection with another actor, known as the respondent

Level

A specified amount of capability for a Sandpiper interaction. Lower levels are less capable and higher levels are more capable

Link

A reference object used by persistent objects to contain multiple relationships and categorization points

Negotiation

Interaction model. Step in which actors establish an exchange and agree to its terms

Node

An individual Sandpiper instance. A human interacting at Level 1-1 technically counts as a node, even if the instance itself is only conceptual (not backed by actual pools of data that can be queried)

Non-Product Data

Information about products that does not define the fundamental nature of a product or its use

PIM

Product Information Manager, used as a shorthand for the original source system of product data

Plan

The agreement between actors

Plan Document

An XML document that unambiguously encapsulates the state of an agreement between actors

Pool

The first division of data within a node, providing owner-level segmentation. There are two types, Canonical and Snapshot

Primary Actor
The node in a transaction sending data, responsible for providing information about and issuing updates from its canonical pools

Product
A single good or service, owned by a creator, with a unique part number

Product Data
The core data that defines a product or its use

Respondent
A Sandpiper actor who accepts contact from another actor, known as the initiator

Secondary Actor
The human, file server, or node in a transaction the receiving data, responsible for providing information about its snapshot pools and processing updates provided by the primary actor

Slice
The subdivision of a Pool and the primary point of connection for Sandpiper exchanges. The slice is the point at which filenames are defined and subscriptions are established

Snapshot Pool
A set of data mirroring the canonical pool from another node, taken at a given point in time

Subscription
A link object allowing a secondary node to subscribe to data changes in a primary node slice

Synchronization
Interaction model step. Transferring product data and resolving pools as part of an exchange between two actors

Transaction
Interaction model. Operations on and communication about data between two actors during synchronization

Copyright Notice

Written by Krister Kittelson, with significant input from the members of the Sandpiper project -- particularly Luke Smith and Doug Winsby.

Licensing

This documentation, part of the Sandpiper Framework software specification and package, is made available to you under the terms of the Artistic License 2.0. The full text of this license follows.

> (b) ensure that installation of your Modified Version does not prevent the user installing or running the Standard Version. In addition, the Modified Version must bear a name that is different from the name of the Standard Version.
>
> (c) allow anyone who receives a copy of the Modified Version to make the Source form of the Modified Version available to others under
>
> > (i) the Original License or
> >
> > (ii) a license that permits the licensee to freely copy, modify and redistribute the Modified Version using the same licensing terms that apply to the copy that the licensee received, and requires that the Source form of the Modified Version, and of any works derived from it, be made freely available in that license fees are prohibited but Distributor Fees are allowed.

Distribution of Compiled Forms of the Standard Version or Modified Versions without the Source

(5) You may Distribute Compiled forms of the Standard Version without the Source, provided that you include complete instructions on how to get the Source of the Standard Version. Such instructions must be valid at the time of your distribution. If these instructions, at any time while you are carrying out such distribution, become invalid, you must provide new instructions on demand or cease further distribution. If you provide valid instructions or cease distribution within thirty days after you become aware that the instructions are invalid, then you do not forfeit any of your rights under this license.

(6) You may Distribute a Modified Version in Compiled form without the Source, provided that you comply with Section 4 with respect to the Source of the Modified Version.

Aggregating or Linking the Package

(7) You may aggregate the Package (either the Standard Version or Modified Version) with other packages and Distribute the resulting aggregation provided that you do not charge a licensing fee for the Package. Distributor Fees are permitted, and licensing fees for other components in the aggregation are permitted. The terms of this license apply to the use and Distribution of the Standard or Modified Versions as included in the aggregation.

(8) You are permitted to link Modified and Standard Versions with other works, to embed the Package in a larger work of your own, or to build stand-alone binary or bytecode versions of applications that include the Package, and Distribute the result without restriction, provided the result does not expose a direct interface to the Package.

Items That are Not Considered Part of a Modified Version

1. In the real world, part numbers can and do overlap between manufacturers. However, only in the case of poorly controlled and structured business data does this happen within the *same* manufacturer. In Sandpiper, the singular part number per creator is a pivotal point and thus required. ↩

2. Some exception may be made for varying "flavors" of things like market copy, usage notes, and so on, but this creates an element of uncertainty that probably discounts these elements from use as anchor points within Sandpiper ↩
3. While a server might run multiple concurrent copies of the Sandpiper software and thus represent multiple nodes, for simplicity we just refer to the node as a system, as this is the most common use case. ↩
4. Splitting an XML file into grains, for example, is highly domain-specific, depending on its format, version, and content. Attempting to natively decode and securely implement this conversion would lead to unacceptable code and scope bloat in an otherwise tightly-specified program. ↩
5. The word "PIM" is ambiguous; it could mean a single program that handles all product information, a combination of systems, or just a human being handling spreadsheets. We use it here as a convenient catch-all term. ↩
6. "Identical in content" specifically means that all meaningful data in the document is the same between the two plans. Differences in non-essential whitespace (leading tabs, spaces, line breaks, etc.) should not be included in this comparison, only the XML elements, attributes, and values. ↩
7. See Wikipedia's entry for more details. Sandpiper specifies version 4 of RFC4122, which describes creating a UUID using a random number ↩