KENETIX™

QUICKSTART GUIDE

globalscape®
unleashing the power of data

GlobalSCAPE, Inc. (GSB)	
	Corporate Headquarters
Address:	4500 Lockhill-Selma Road, Suite 150, San Antonio, TX (USA) 78249
Sales:	(210) 308-8267
Sales (Toll Free):	(800) 290-5054
Technical Support:	(210) 366-3993

Web Support: http://www.globalscape.com/support/

June 21, 2017

Table of Contents

Introduction to Kenetix

We are all connected. With the advent of computer networks, we are now a global community. The world of connected devices is no longer an eventuality, it is reality. Mass quantities of information are being propagated across web-enabled devices. Keeping up with this influx of new information is becoming nearly impossible.

Where there is difficulty, there is often opportunity. Companies, large and small, understand the value of tapping into this stream, but are often prohibited by both the complexity required to access this information and to turn this information into something actionable. Understanding how data flows, how it moves between devices, and how it affects both our physical and non-physical world is soon going to be a crucial skill.

What is Kenetix?

Kenetix is a lot of different things. Kenetix is a tool for workplace automation, a platform for integrating cloud applications, a visual programming language, and more.

Across industries, businesses are moving their services to the cloud. Gone is the day of Service-in-a-Silo. Having a product that is connected to the cloud is analogous to having a business address. People need to know where, and how, to reach you.

Businesses react to the demand to be found by throwing open their doors. They create interfaces for interacting with their products programmatically. These interfaces are gateways for retrieving or manipulating data within their service. This is the next advent of networked computing, the ubiquitous sharing of resources between separate services.

Kenetix allows you to leverage all of these emerging resources by giving you capable, cloud-based tools. It allows you to automate workflows, create microservices, and develop cloud applications all within a single, powerful design environment.

To learn more about Kenetix, keep reading this guide. As with many learning resources, each chapter builds upon the other. However, if you're only interested in learning about a specific topic (such as Types) feel free to skip ahead to the other chapters.

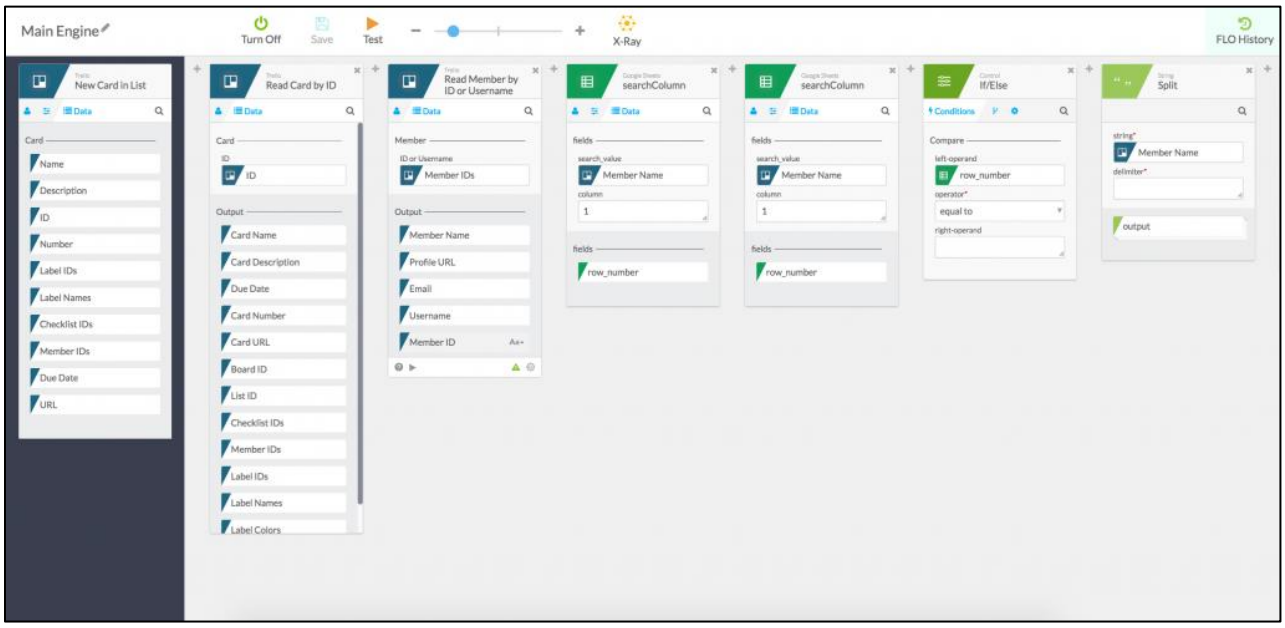
1. Visual Development: The Language of Kenetix

Designer is the core tool for building applications in Kenetix. Applications built in Kenetix are event-driven.

You can think of the elements inside Designer as the language of Kenetix. We'll go into more depth on what each one of these elements are later, but here's the basic list of Designer elements:

- **FLO**: an application container inside Designer
- **Card**: a unit of functionality, either an Action or Event
- **Input** Field: a field that accepts a value of a certain type that is user defined
- **Output Field**: a field that contains a value that is defined at the runtime of the FLO

Everything you build inside Designer contains these elements. You create FLOs, which contain Cards. Each Card accepts data as an input and returns new data as an output.



Kenetix offers a library of prebuilt cloud applications that allow you to activate a FLO when events occur inside a selected cloud application.

It is also possible to expose a FLO as an API endpoint. This allows external services to activate the logic contained within the body of your FLO.

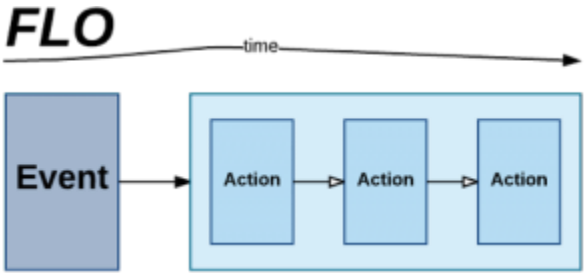
This resource will teach you the ins-and-outs of creating integrations with Kenetix. Learning how to use Kenetix to integrate cloud apps is incredibly powerful and useful. This resource, though, will also teach you how to build complex applications using Kenetix.

2. FLO Structure: How to Build a Flo

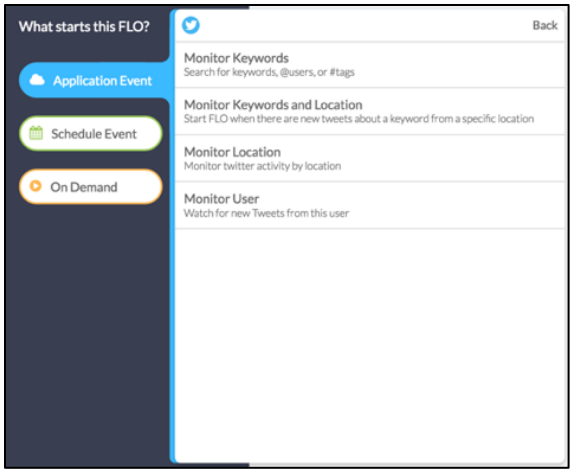
A FLO is a series of Cards. In a FLO, the first Card will always be an **Event**. An Event must always precede **Actions**. In any FLO, there is only one Event Card.

Every Card after an Event is an Action, which defines what your FLO does.

When an Event occurs within your FLO, the FLO performs actions, one by one, from left to right.



There are two types of Actions within Kenetix: **Application Actions**, which pertain to the specific cloud application you are working with, and **Function Actions**, which allow you to control or manipulate data being returned from your Application Actions.



Working with Applications

The way Kenetix works with your applications is through Events and Actions, such as Gmail, Salesforce, or Smartsheets.

An application may contain many different unique Events and Actions. You can use these unique Events and Actions to build FLOs.

Events

There are three types of events: **Application Events**, **Scheduled Events**, and **On Demand Events**. When a FLO will execute depends on the Event specified.

With an Application Event, you can monitor for changes within your cloud application. When a change has been detected, the FLO will execute.

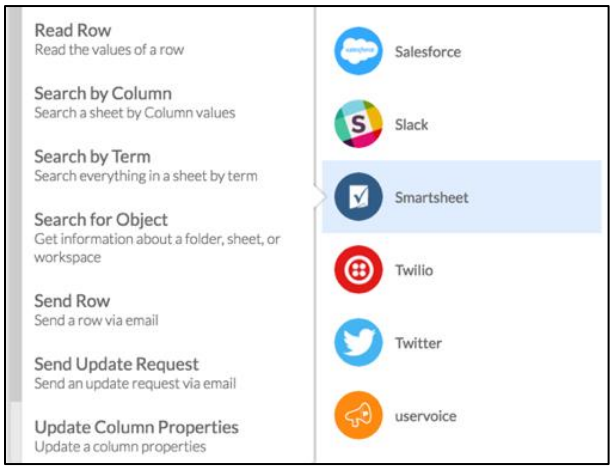
Scheduled Events allow you to execute a FLO at a given interval.

On Demand Events allow you to activate a FLO internally or externally, depending on your needs.

Examples of available Events:

- **Application**: When a new Customer is created in Salesforce,
- **Scheduled**: Every hour on the hour, execute this FLO,
- **On Demand**: Execute this FLO after a customer has filled out a web form.

Actions



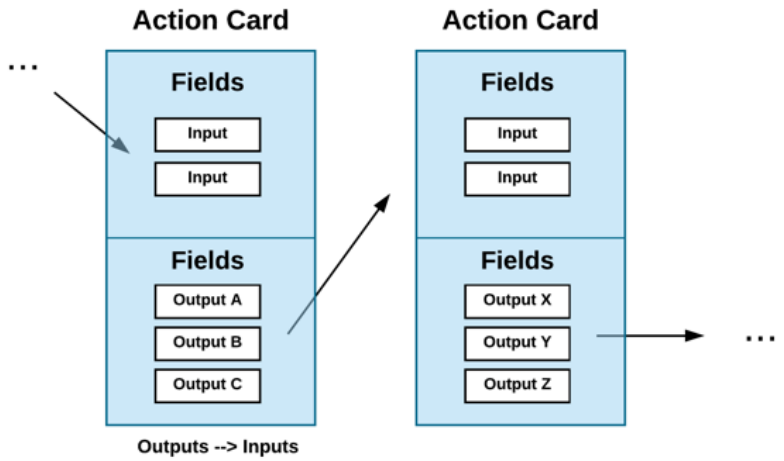
Actions allow the FLO to tell the cloud application to do something. Examples of Actions include:

- Send an email through Gmail,
- Update a contact in Salesforce,
- Modify a row in Smartsheet.

Each application has a unique set of Events and Actions, according to the capabilities of the cloud application.

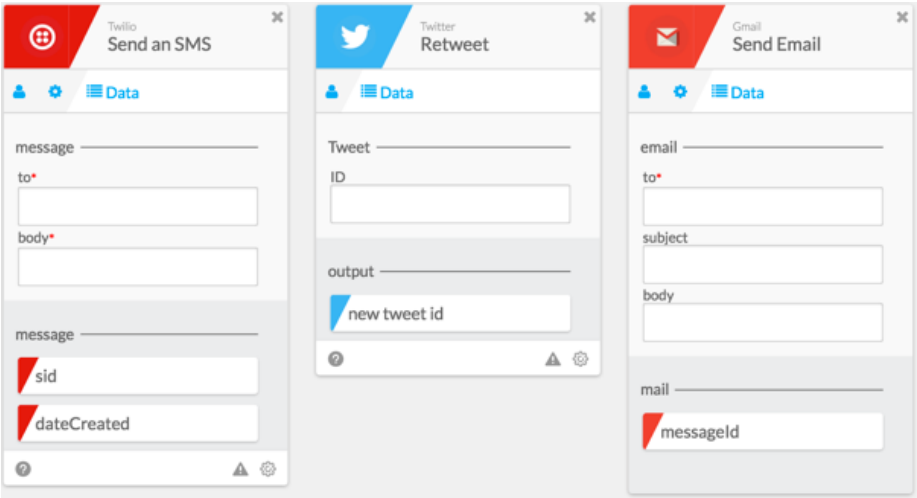
Cards -- Get More "Out" Than You Put "In"

The ability for an Action Card to accept input, and then return a different output, is one of the most powerful features of Kenetix:



Each Card represents one specific step in a FLO. Within a Card, there are two types of fields: input and output. **Input** fields accept values. **Output** fields are values that a Card produces that can be used by subsequent Cards.

Here are a few examples of Action Cards from Kenetix:



Each of these three Cards is an Action from a specific application. In this case, it's an Action Card from the Twilio application, an Action Card from the Twitter application, and an Action Card from the Gmail application.

The Twilio Action Card contains two input fields: ***to,*** and ***body***. It also contains two output fields: ***sid*** and ***dateCreated***.

Let's Get Serious with FLOs and Functions

Remember: FLOs are just Actions and Events. However, there are two categories of Actions: **Application Actions** and **Function Actions**.

We've covered Application Actions, but we've yet to touch on the topic of Functions.

A Function is a Card that allows you to interact with, change, or control the data in your FLO. Typically, a Function Card is placed in-between Action Cards.

Categories of Function Cards include (but are not limited to):

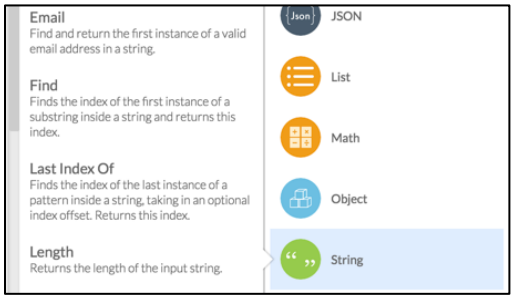
- **Boolean**: evaluate values based on true or false conditions,
- **Control**: manage and manipulate the structure of your FLO,
- **Crypto**: encrypt or decrypt data,
- **Date**: parse and manipulate times and dates,
- **List**: create and iterate over lists of items,
- **Math**: perform mathematical operations,
- **String**: build, modify, and parse text.

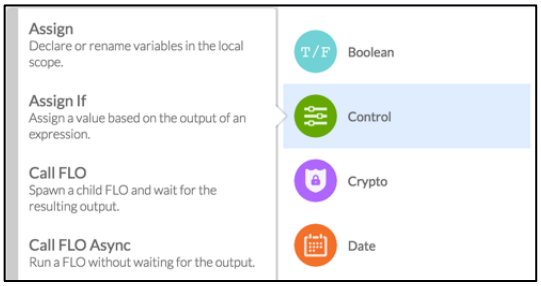
Example: a simple FLO that uses Functions might use a Card within the Control category of Functions, Continue If, to determine whether or not a FLO should continue, based on a specific condition.

Or, the function **Compose** within the **String** category, which allows you to compose a new message that contains output fields from other Cards.

Functions such as these can be chained together to create complex interactions between your Cards, providing a useful paradigm that allows you to create powerful workflows.

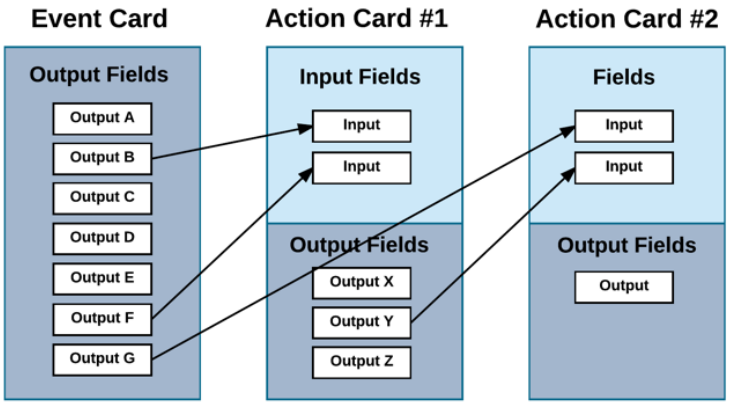
Below are examples of Functions contained within each respective category:





Bringing It All Together

The diagram below represents almost all the building blocks necessary to create complex FLOs.



Let's unpack this diagram, step-by-step:

1. An Event Card from a specific application generates a set of outputs, Output A through Output G.
2. Output B and Output F from the Event Card are used as inputs for Action Card #1. As a result, Action Card #1 generates Output X, Y, and Z.
3. Output G from the Event Card, and Output Y from Action Card #1, are used as inputs to Action Card #2.

Theory Is Great, Practice Is Better

Every element inside of a FLO is a Card. Every Card is one of three things: Event or Action. When thinking about how to design a FLO, it's useful to remember this.

FLOs are just series of inputs and outputs. Something goes in, something comes out.

In the next chapter, we're going to cover Types inside Designer. This is a fundamental concept that will help you understand how data is manipulated and moved during its lifecycle.

3. Types

All input and output fields in the Designer have a type. There are five basic types in Designer: **String**, **Number**, **True/False**, **Date**, and **File**.

When working with input fields and output fields, it's important to be mindful of the types. The type system is what allows or prevents certain behaviors.

- **String**: A sequence of characters, wrapped in either single quotes or double quotes.

  Example:

  ```
  'Free Mars', "Free Mars", '12345', "12345", "", " "
  ```

  In the last two examples, "" represents an empty string, and " " is a string that contains one character, an empty space.

- **Number**: An integer or decimal.

  Example:

  ```
  12345, 1, -10, 0.543, 500000
  ```

- **True/False**: Boolean values true and false. These values are not considered strings. However, both Number and String types can convert to the True/False type within the Designer.

  For example, the string 'true' and the string 'false' will both work as input to a True/False input field. The Designer will automatically convert these values to their respective Boolean values. Additionally, any non-empty string values will also evaluate to True if used in a True/False input field. Empty strings, however, evaluate to False.

  Similarly, the integer 0 and the integer 1 will also work as inputs to True/False. 0 is representative of the Boolean value false, and 1 is representative of the Boolean value true.

Special Types

Beyond these basic types, there are also two special types, List and Object. These types are composites of the other types.

A list is a collection of items all of the same type. An object is a collection of keys and values, where each value is of a certain type. Unlike lists, objects can contain values of many different types.

A list has the structure:

```
[ 'nitrogen', 'oxygen', 'argon' ],
```

where 'nitrogen', 'oxygen', and 'argon' represent three String items in a list.

An object has the structure:

```
{"lastName": "Russell", "firstName": "Saxifrage"}
```

In the above example, "lastName" and "firstName" are the keys in the object. The corresponding values are "Russell" and "Saxifrage".

Items in a list are often represented as contained between brackets **[]**. Similarly, you can distinguish an object by its use of curly braces **{}**.

The Right Types

Type mismatches occur when the Designer expects a certain type but does not receive it. Most of the time, the Designer will prevent you from making these sorts of errors. However, they can occur with more advanced use of the Designer, so it's important to be mindful of the overall type system.

Both input fields and output fields have a type. However, input fields and output fields both handle types differently.

Input Fields

For an input field, if a certain type is specified, the Card will try to automatically convert an incoming value to that type. For example, if you were to use the value "4", a String, (note the double quotes) as input to a field that was set to the type Number, the Card will automatically convert this value to the appropriate type.

It will then be considered the Number 4 and be able to be used as such.

Output Fields

Output fields, however, behave differently. Output fields expect you, the user, to be honest about what they are. If you specify that an output field is of the type "String", the Designer will expect that the value contained within this output field is a string. In the circumstance that the output field value does not match its type, a type conversion error will be thrown.

Many of the available functions in the Designer already specify both the type of the input field and the output field and do not allow you to change these types. The Designer will often know what type both the inputs and outputs should be. In this circumstance, worrying about types is unnecessary.

However, in circumstances where the Designer doesn't explicitly know either the input or the output field values, types play a large role.

4. Math and Strings: Learning Functions

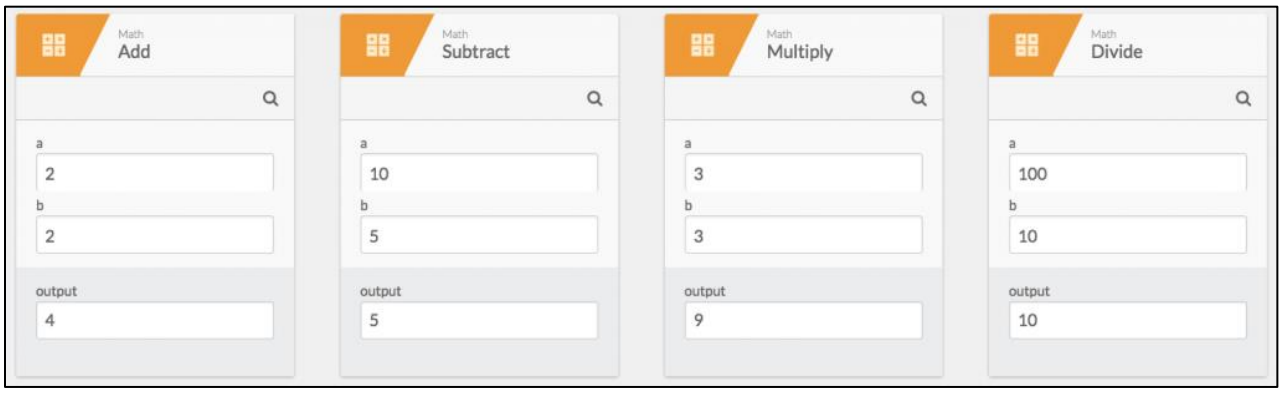
In many programming languages, functions are defined as "procedures or routines that accept an input and return a new output." With Kenetix, this same definition applies. There are a number of different Function categories, and in each category, there are Function Cards that provide different capabilities.

The categories of functions that are currently available in Designer are: Boolean, Control, Crypto, Date, HTTP, JSON, JWT, List, Math, Object, String, Text Analysis, URL, and XML.

It's not necessary to memorize every function inside Designer. However, it is good to have a base familiarity with some common functions, and (of course) know how to use functions most effectively. Math and String functions are two of the more common libraries that are used when building a FLO, so in this chapter, we'll explain functions through that lens.

Math Functions

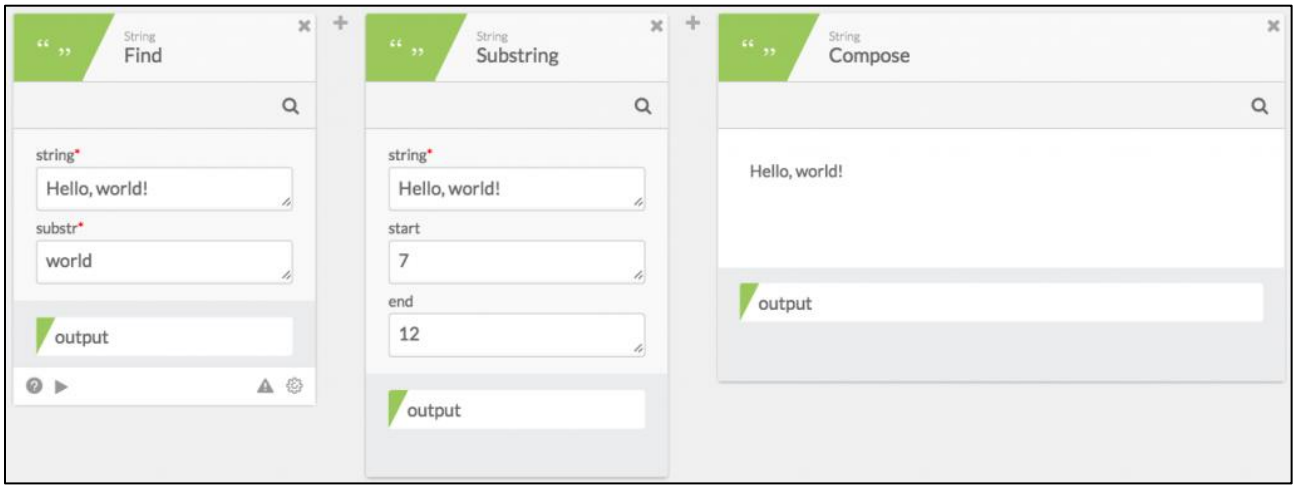
Math functions are used to perform common Math operations, such as Add, Subtract, Multiply, and Divide.



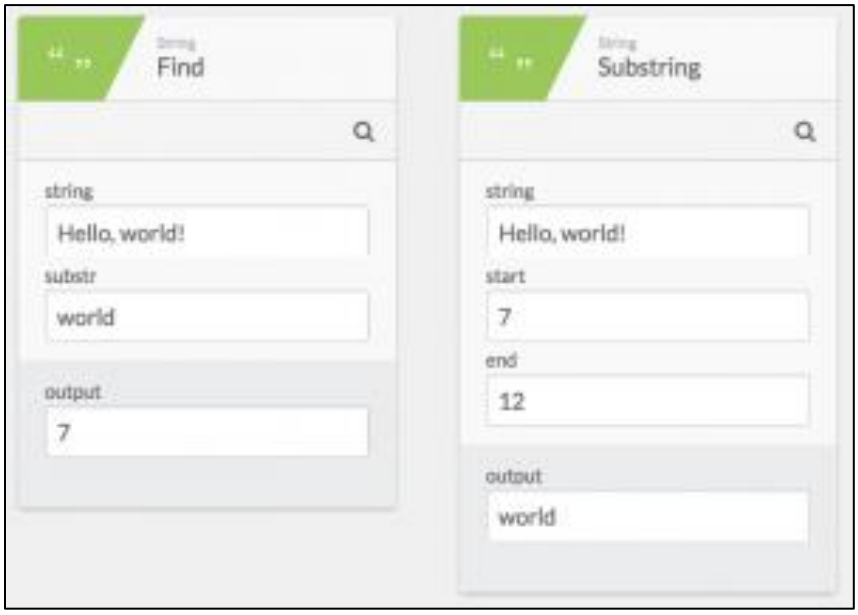
In addition to these basic functions, there are other more complex functions available, such as Factorial, Modulo, and Ceiling.

String Functions

There are a number of different String functions available for your use, but three of the most important string functions are Find, Substring, and Compose.



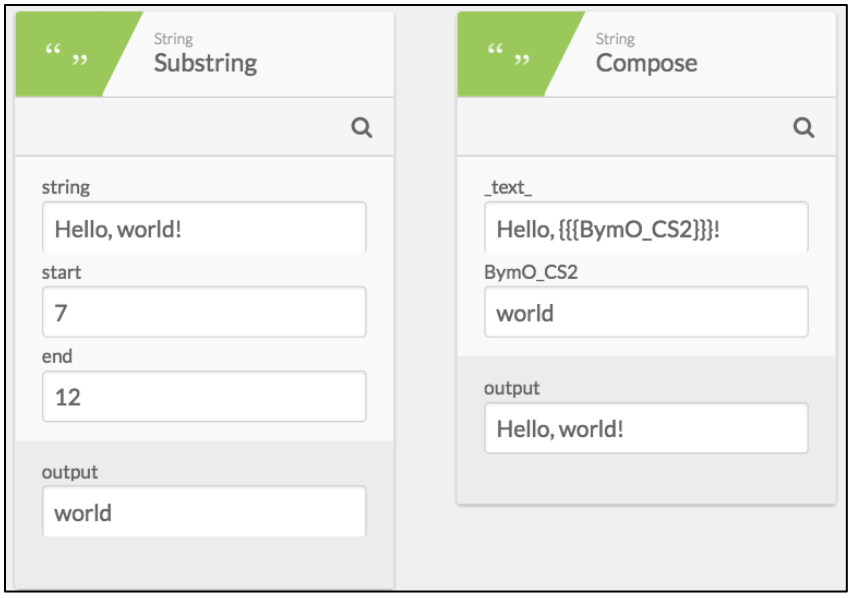
Find allows you to search for a string inside another string. This can be useful if you're looking for a specific string (this function will return an output field of "-1" if a string is not found). If the string you're trying to find exists, the index of the first character of the string will be returned as the output.



This function goes hand-in-hand with our next function, Substring. Substring allows you to extract a value from a string by specifying the starting location of the string and the ending location of the string. These locations are referred to as the 'starting index' and the 'ending index'.

You can think of Substring as grabbing only a very specific section of a string. This may be a word, a character, or a series of words and characters. With Substring, you are grabbing a slice of another string.

Our last function, Compose, is one of the most useful string functions. It allows you to drag-and-drop output fields into a textbox and "compose" new strings from these fields. This is functionally similar to the Concatenate function that is also available.



In the above example, the Substring Card is used to extract the word "world" from the string. This output is then supplied to the Compose Card, allowing you to concatenate together the two strings. While it's hard to tell, the input field with the label 'BymO_CS2' is actually a reference to the 'world' output field from the Substring Card.

These are examples of common Cards or patterns you might find yourself needing to use while designing FLOs. Think of the functions inside Designer as the tools in your toolbox. For every job, there is the right tool!

5. Lists and Loops

Inside Designer, lists are represented as three output fields stacked diagonally.

When working with lists, the position of an item in a list is referred to as its index.

In Chapter 3, we were introduced to the list:

```
['nitrogen', 'oxygen', 'argon']
```

If you were to number these items, you might expect that 'nitrogen' would be at index (position) 1 in the list, 'oxygen' at index 2, and 'argon' at index 3. However, this is not true! When counting the number of items in a list, we don't begin at 1, we begin at 0. If we look back to our previous list, then, the item 'nitrogen' is actually at index 0 in our list, 'oxygen' is at index 1, and 'carbon' is at index 2.

Remember: a list always begins at index 0. This is often referred to as zero-based array in other programming languages.

Types of Lists

In the previous example, we had the list:

```
['nitrogen', 'oxygen', 'argon'],
```

This list contains three items. Each item is of the type String. Items that are type String are always wrapped in either single quotes or double quotes. Since we know that each item in the list is a string, we know that this is a list of Strings.

Every item in this list is a String, and any subsequent items added to this list must also be of the type String.

Lists can contain items of the type Number, String, Date, True/False, and File.

Examples:

List - Number

```
[2, 4, 6, 8]
```

List - Object

```
[{"firstName": "Carly Rae", "lastName": "Jepsen"}, {"firstName": "Taylor",
"lastName": "Swift"}]
```

Working with Lists

Lists are a collection of items of a specific type. Unlike single items, lists often require the use of List functions. List functions allow you to access and manipulate items in a list. There are a number of functions that allow you to interact with a single item in a list, the most common being **List - At**.

If you want to work with an entire list of items, you must iterate over each item in succession until a specific condition is met, or the end of the list is reached.

This process of iteration is called **looping** and is one of the core concepts behind list. When you loop through a list, you execute a series of statements (that you specify) per item in a list. This means that if you have 5 items in a list, the statements that you define will be executed 5 times.

Often, you will use this functionality to 'process' each item in a list, using the item in the context of the loop body. The body of a loop is just a series of statements. Statements, in the context of Designer, are Cards.

This allows you to create a unique sequence of steps that can be used repeatedly regardless of the items passed in. This is very useful when processing large lists of items.

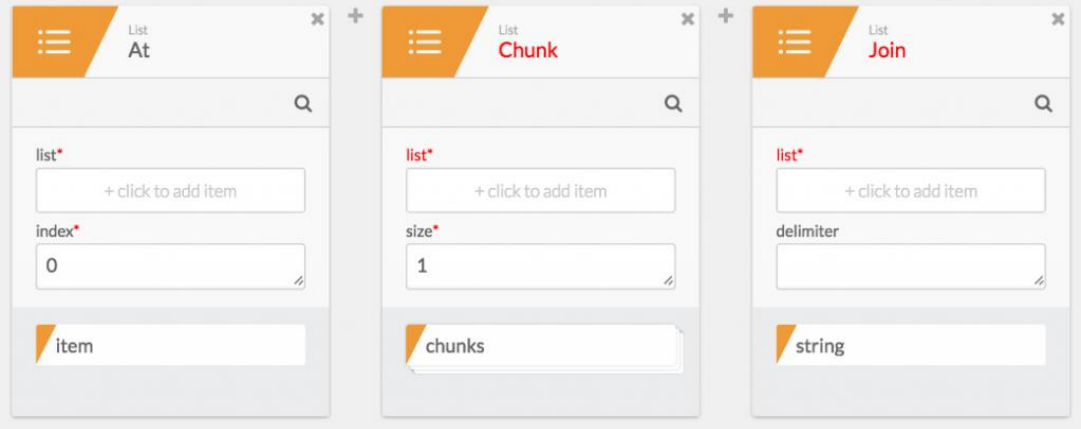
Below are some of the core functions for iterating over a list:

- Filter, Filter By: Filter a list based on conditions.
- Find, Find By: Find a specific item in a list.
- For Each: Iterate over the entirety of the list, executing a set of statements per each iteration.
- Map: Take an existing list, execute statements, and return resulting list.
- Reduce: Reduces a list down to one item based on a set of conditions.

This is not the exhaustive list of available List functions, but rather a subset of useful functions for iterating over a list.

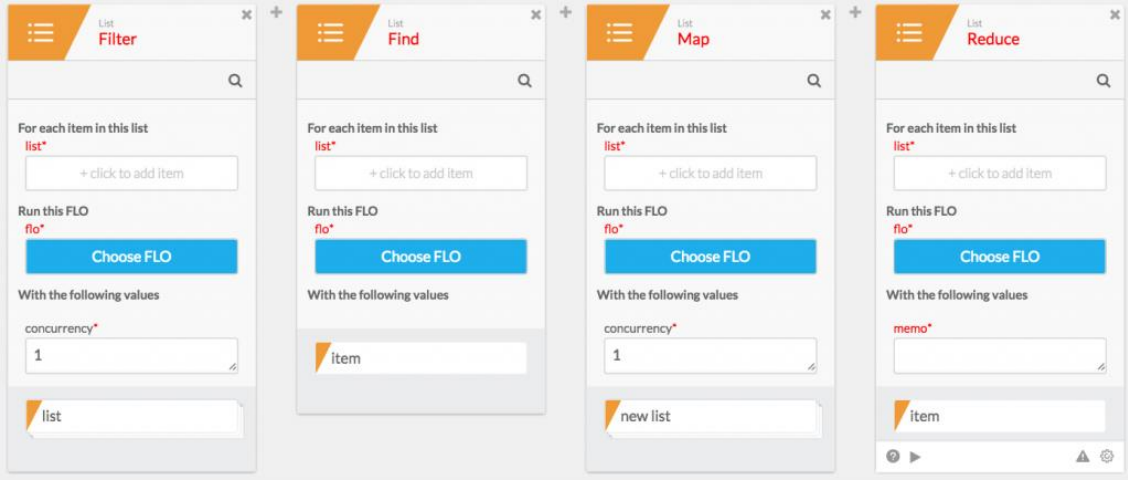
When working with lists, there are two types of List function Cards.

The first type of Card accepts as an input a list and returns an output. These Cards may require other input field values, as shown below:



The second type of Card accepts as an input a list, and then requires that you specify a set of statements/instructions beforehand that execute per iteration of item in the list.

Filter, Find, Map, and Reduce are examples of this type of Card:



Each of these Cards requires you provide a FLO that contains a set of statements that will be executed on each iteration.

A good rule of thumb when using List functions Cards that require iteration is to have previously created the FLO that contains the statements to be executed.

Lists are a useful construct when working with collections of items.

6. Objects

In the previous chapters, we lightly introduced the idea of objects. In this chapter, we're going to take a closer examination of what objects are and how they are used in Designer.

What are Objects?

Objects are collections of properties. A property is a key-value pair that allows you to associate a specific value with a key. For example, the following object has two properties:

```
{"lastName": "Russell", "firstName": "Saxifrage"}
```

The two keys, lastName and firstName, point to each of their respective values. In this example, "lastName" is the key for the value "Russell", and "firstName" is the key for the value "Saxifrage".

All values in objects are referenced by their key.

An object can contain any number of properties. Properties inside an object don't have to have the same type as the other properties, meaning you can have an object that consists of Strings, Numbers, True/False, File, and Date types.

Example:

```
{"lastName": "Russell", "firstName": "Saxifrage", "age": 93, "isAlive": true,
"lastSeen": "2016-09-20T19:01:21.802Z"}
```

The above examples are written in JavaScript Object Notation (JSON).

A JSON object is always wrapped in curly braces. Properties inside the object are enclosed in double quotes. The value associated with the key can be of any type. Properties inside an object are delimited by commas.

Using Object Functions

Inside Designer, the Object function Cards can be used to interact with objects. Basic tasks, such as retrieving values or setting values on an object, can be achieved by using Cards such as Object - Get, Object - Pick, and Object - Set.

It's also possible to construct your own object by manually inputting value or dragging-and-dropping output fields into an Object - Construct Card.

Objects are a useful data type in that they provide an easy way to make key-value associations. Whenever you need to assign a key to a specific value, you can use an object to do so. Objects and Lists are the two data types that can encapsulate data, so it can be moved and worked with easily.

Our previous example illustrates the usefulness of objects:

```
{"lastName": "Russell", "firstName": "Saxifrage", "age": 93, "isAlive": true,
"lastSeen": "2016-09-20T19:01:21.802Z"}
```

By containing all of this information in a single object, you are able to create a context. Of course, objects can be used to store information about anything you want. You create the context for the object.

The "context" of an object can also be described as its schema. A schema is a blueprint describing what will be contained within the object.

Using objects as a data structure in this way is very powerful. Many different web services leverage objects, and JSON in particular, to deliver information to other services via their API. This paradigm can be applied in many different areas.

7. Building a Connector

The connector builder is the tool inside Kenetix used to create connectors. A connector is an integration to a specific service (cloud, on premises) that can be defined in terms of actions and events. You use the connector builder to create collections of actions and events that are collectively referred to as a connector.

You have two distinct environments in the connector builder: the workspace and the control panel. When building a connector, you'll be using the workspace. Once your connector is finished, you can deploy the finished connector to your organization by using the Control Panel.

At its most basic, you construct connectors in the connector builder by providing the appropriate values to JSON key/value pairs. You can conceptualize these JSON key/value pairs as instructions that define the desired the functionality of the connector. These instructions are executed sequentially, resulting in the build of the connector.

When creating a connector for the first time, the initial JSON required to construct the connector will be generated for you. You can use these generated templates by filling out values to the required keys. Most things inside the connector builder follow this same structure. Modules, authentications, helper functions, and beyond, are just templates that you selectively choose and input value, depending on desired functionality.

For example, when creating a connector that leverages OAuth2 for authentication, JSON will be generated to support that appropriate type of authentication. You will need to supply input values (such as tokens, secrets, authorization paths, etc.) to appropriately configure authentication.

Understanding that the connector builder is just executing a series of instructions, instructions that are often generated automatically and then require certain input from you, can help you better conceptualize how to use the connector builder.

8. What is a Connector?

A Connector is an interface that communicates with external APIs. The goal of a Connector is not to be a direct reflection of an API, but instead a user-friendly abstraction on top of an API. Connector development is more focused on end-user needs than on the capabilities of the API. Each Connector is made up of methods, which appear to the user as the different Event and Action Cards. Each method determines how data is fetched from the API, and transforms data to the user-friendly format accepted by the front-end. Most methods use more than one API call to do this.

To define a Connector, you must write a Connector JSON file and submit this file to Kenetix for upload into the engine. At runtime, the engine will access the instructions laid out in this file to execute the Event or Action the user has designated in their FLO. The primary function of a Connector JSON file is to lay out in a linearly the pre-defined action steps (known as modules) that will execute each user-facing event or action. Using modules, you can define what data will be fetched from the API, select the data you want to pass on to the user, and transform this data into flat JSON that can be consumed by the Kenetix front-end.

Each Connector has 4 sections:

The **Authentication** section, where you define the Authentication schema of the Connector. Kenetix supports 3 authentication schemes: basic, OAuth (both 1 and 2) and custom. Once users have set up an authentication configuration for a Connector, Kenetix will save their credentials so this configuration can be used again.

The **Events** section, where you define any Event Cards you want in your Connector. Events are the methods that start FLOs. Each FLO must have a starting Event, and every time the Event detects new records the FLO will run.

The **Actions** section, where you define any Action Cards you want in your Connector. Actions are the methods that make up the majority of a FLO, and execute whenever the starting Event occurs.

The **Functions** section, where you define any helper methods you need to execute your Events and Actions. While Events and Actions are turned into Cards that are visible to the user, helper methods are never seen by the user and always execute under the hood.

9. Connectors and Modules: A Brief Overview

Connectors are built in the connector builder as a series of sequential instructions, or steps. Each step requires that you use a module to drive a certain functionality. You can understand modules to be functions that provide helpful utility in relation to building a connector. These modules let you manipulate objects and strings, iterate over collections of items, perform complex calculations, and more.

Each step in the connector builder is just a module performing a specific function. You must use modules to drive the functionality of the connector.

A connector, then, will end up just being a series of modules, strung together in succession. Often, you may use modules such as "HTTP - Get", which allows you to request data from an HTTP resource, or "List - Map" which allows you to take data from a source and map it to a new output based on specific criteria. You can find out what modules are available inside the Modules Reference.

Passing Data Between the Modules

Each module has a field wherein you're allowed to specify a unique identifier. A unique identifier is generated automatically, taking the form of a short string e.g., "RjKlX", but you can set the id of a module to whatever unique value you like. The outputs that the module generates can be referenced in other modules by using this id.

Referencing Output Data from Modules

Inside the connector builder, we use Mustache to help handle references to data. Let's say that we are using an HTTP - Get module, and we have specified that the id of the module is going to be "getGoogleMapsData". This module returns a response, and inside the response there is a body object that contains data. This data can be accessed as output values that are able to be referenced by Mustache.

You would access the data contained inside the body object by using the following syntax:

```
"{{getGoogleMapsData.body.location}}"
```

where "getGoogleMapsData" is the ID associated with the module, "body" is an object, and "location" is a single field/property inside this object.

Other examples:

```
"{{getGoogleMapsData.body.latitude}}"
"{{getGoogleMapsData.body.longitude}}"
"{{getGoogleMapsData.body}}"
```

You can identify the names of these fields by examining the response that is returned by the module, using the connector builder console.