

OpenWhisk Package Specification

Version 0.8, Working Draft 09, Revision 1

Notational Conventions

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in [RFC 2119](#).

The OpenWhisk specification is licensed under [The Apache License, Version 2.0](#).

Introduction

OpenWhisk™ is an open source, distributed Serverless computing project. Specifically, it is able to execute application logic (*Actions*) in response to events (*Triggers*) from external sources (*Feeds*) governed by simple conditional logic (*Rules*) around the event data.

It provides a programming model for registering and managing *Actions*, *Triggers* and *Rules* supported by a REST-based Command Line Interface (CLI) along with tooling to support packaging and catalog services.

The project includes a catalog of built-in system and utility *Actions* and *Feeds*, along with a robust set of samples that demonstrate how to integrate OpenWhisk with various external service providers (e.g., GitHub, Slack, etc.) along with several platform and run-time Software Development Kits (SDKs).

The code for the *Actions*, along with any support services implementing *Feeds*, are packaged according to this specification to be compatible with the OpenWhisk catalog and its tooling. It also serves as a means for architects and developers to model OpenWhisk package *Actions* as part of full, event-driven services and applications providing the necessary information for artifact and data type validation along with package management operations.

Compatibility

This specification is intended to be compatible with the following specifications:

- *OpenWhisk API which is defined as an OpenAPI document:*
 - <https://raw.githubusercontent.com/openwhisk/openwhisk/master/core/controller/src/main/resources/whiskswagger.json>
- *OpenAPI Specification when defining REST APIs and parameters:*
 - <https://github.com/OAI/OpenAPI-Specification/blob/master/versions/2.0.md>

34 Revision History

Version	Date	Notes
0.8.1	2016-11-03	Initial public point draft, Working Draft 01
0.8.2	2016-12-12	Working Draft 02, Add. Use cases, examples
0.8.3	2017-02-02	Working Draft 03, Add use cases, examples, \$ notation
0.8.4	2017-04-18	Working Draft 04, Support JSON parameter type; Clarify use of Parameter single-line grammar and inferred types. Add support for API Gateway mappings. Add support for Web Actions
0.8.5	2017-04-21	Add support for “dependencies”, that is allow automatic deployment of other OpenWhisk packages (from GitHub) that the current package declares as a dependency.
0.8.6	2017-07-25	<ul style="list-style-type: none"> • Clarified requirements for \$ dollar notation. • Updated conceptual Manifest/Deployment File processing images.
0.8.7	2017-08-24	<ul style="list-style-type: none"> • Added explicit Application entity and grammar. • Added API listing to Package entity. • Cleaned up pseudo-grammar which contained various uses of credentials in places not intended. • Fixed Polygon Tracking example (indentation incorrect).
0.8.8	2017-08-29	<ul style="list-style-type: none"> • Created a simplified API entity (i.e., “api”) grammar that allows multiple sets of named APIs for the same basepath. • Acknowledge PHP as supported runtime (kind). • Added “sequences” entity as a convenient way to declare action sequences in the manifest. Updated supported runtime values.
0.8.9 0.8.9.1	2017-09-22 2017-09-29	<ul style="list-style-type: none"> • Clarified “version” key requirements for Package (required) and Action (optional); removed from shared entity schema. • Made “license” key optional for package. • keyword “package” (singular) and “packages” (plural) both allowed. • Adjusted use case examples to reflect these changes. • Rework of schema use cases into full, step-by-step examples. • Spellcheck, fixed bugs, update examples to match web-based version.

36	Table of Contents
37	OpenWhisk Package Specification 1
38	Version 0.8, Working Draft 09 1
39	<i>Introduction 1</i>
40	Compatibility 1
41	Revision History 2
42	<i>Table of Contents 3</i>
43	<i>Programming Model 5</i>
44	OpenWhisk Entities 5
45	Cardinality 5
46	Conceptual representation 6
47	<i>Package processing 6</i>
48	Conceptual Package creation and publishing 6
49	Conceptual tooling integration and deployment 7
50	Composition 8
51	Namespacing 8
52	Entity Names 9
53	Definitions 9
54	<i>Specification 9</i>
55	YAML Types 10
56	OpenWhisk Types 10
57	Schema 12
58	Extended Schema 29
59	Package Artifacts 33
60	Normative References 36
61	<i>Non-normative References 36</i>
62	Scenarios and Use cases 37
63	<i>Usage Scenarios 37</i>
64	Guided examples 39
65	<i>Package Examples 39</i>
66	Example 1: Minimal valid Package Manifest 39
67	<i>Actions Examples 39</i>
68	Example 1: The “Hello world” Action 39
69	Example 2: Adding fixed Input values to an Action 41
70	Example 3: “Hello world” with typed input and output parameters 41
71	Example 4: “Hello world” with advanced parameters 43
72	Example 5: Adding a Trigger and Rule to “hello world” 44
73	Example 6: Using a Deployment file to bind Trigger parameters 46
74	Github feed 47
75	<i>Advanced examples 48</i>
76	Github feed advanced 48
77	RSS Package 49
78	Polygon Tracking 50
79	MQTT Package (tailored for Watson IoT) 53
80	Check deposit processing with optical character recognition 55
81	Event Sources 60
82	<i>Curated Feeds 60</i>

83	Alarms 60
84	Cloudant 61
85	<i>Public Sources 61</i>
86	GitHub WebHook 61
87	Other Considerations 62
88	<i>Tooling interaction 62</i>
89	Using package manifest directly from GitHub 62
90	Using package manifest in archive (e.g., ZIP) file 62
91	<i>Simplification of WebHook Integration 62</i>
92	Using RESTify 62
93	<i>Enablement of Debugging for DevOps 62</i>
94	Isolating and debugging “bad” Actions using (local) Docker 62
95	Using software debugging (LLDB) frameworks 62
96	Acknowledgements 63
97	

98 Programming Model

99 OpenWhisk Entities

100 OpenWhisk uses the following entities to describe its programming model:

101 *Action*

102 A stateless, relatively short-running function (*on the order of seconds or even milliseconds*) invoked as an
103 event handler.

104 *Trigger*

105 The name for a class of events. Triggers represent the events (and their data) themselves without any
106 concept of how they were generated.

107 *Rule*

108 A mapping from a Trigger to an Action which may contain simple conditional logic. OpenWhisk
109 evaluates incoming events (that belong to a Trigger) and invokes the assigned Action (event handler).

110 *Event Source*

111 An Event Source is the descriptor (edge) for an Event Producer (or provider). It describes the Event
112 Format(s) produced, as well as any configuration and subscription capabilities.

113 *Feed*

114 A Feed is an optional service that represents and controls the stream which all belong to a Trigger. A feed
115 provides operations called **feed actions** which handle creating, deleting, pausing, and resuming the stream
116 of events. The feed action typically interacts with external services which produce the events

117 *Package*

118 A named, shared collection of Actions and Feeds. The goal of this specification is to describe OpenWhisk
119 packages and their component entities and resources to enable an open-ecosystem.

120
121 *Packages are designed to be first-class entities within the OpenWhisk platform to be used by tooling such*
122 *as catalogs (repositories), associated package managers, installers, etc.*

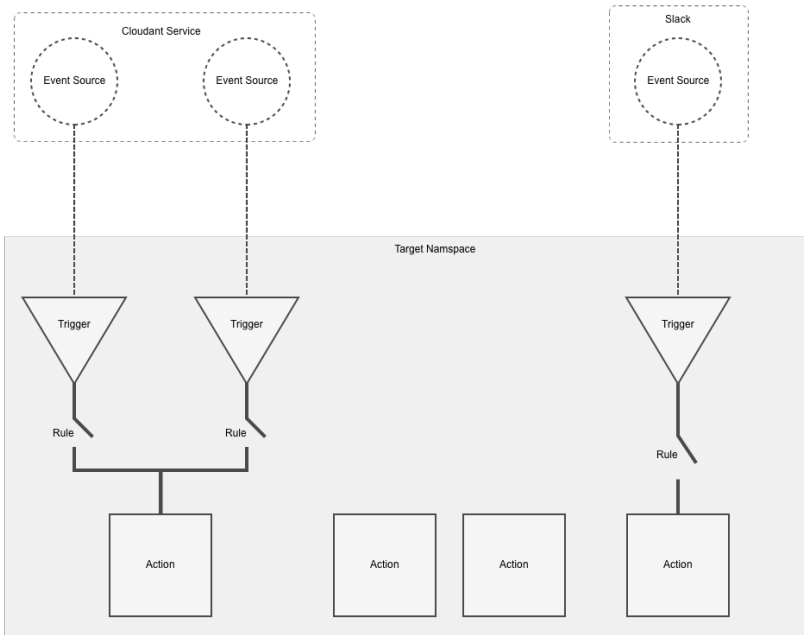
123
124 *Note: Not all actions must belong to packages, but can exist under a namespace.*

125 Cardinality

126 *Trigger to Action*

127 With the appropriate set of Rules, it's possible for a single Trigger (event) to invoke multiple Actions, or
128 for an Action to be invoked as a response to events from multiple Triggers.

Conceptual representation



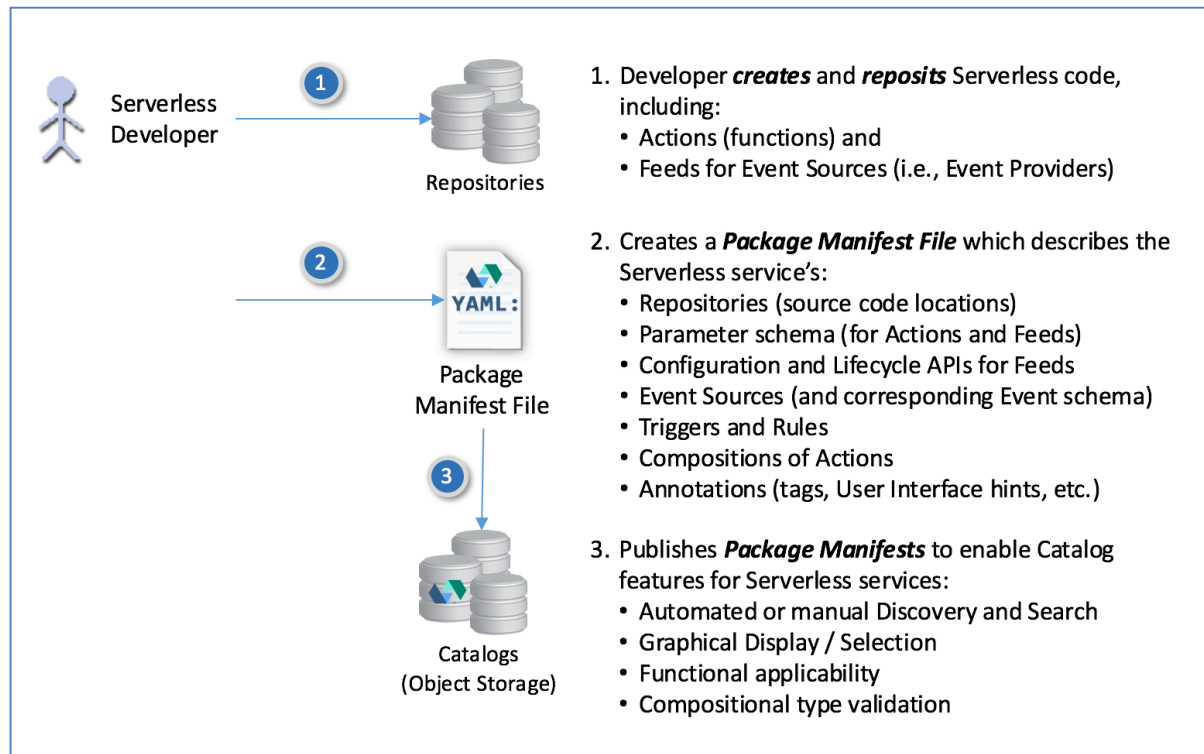
Package processing

This document defines two file artifacts that are used to deploy Packages to a target OpenWhisk platform; these include:

- **Package Manifest file**: Contains the Package definition along with any included Action, Trigger or Rule definitions that comprise the package. This file includes the schema of input and output data to each entity for validation purposes.
- **Deployment file**: Contains the values and bindings used configure a Package to a target OpenWhisk platform provider's environment and supply input parameter values for Packages, Actions and Triggers. This can include Namespace bindings, security and policy information.

Conceptual Package creation and publishing

The following diagram illustrates how a developer would create OpenWhisk code artifacts and associate a Package Manifest file that describes them for deployment and reuse.

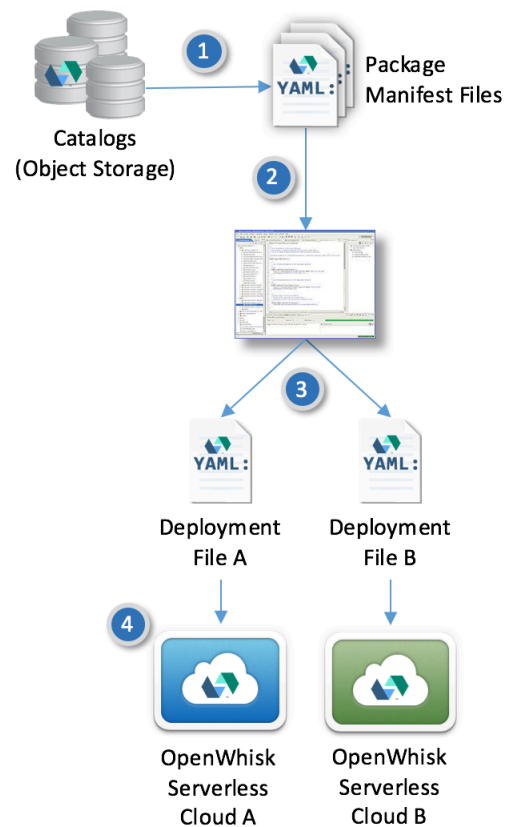


143

144 Conceptual tooling integration and deployment

145 The following diagram illustrates how Package manifests can be leveraged by developer tooling to
 146 integrate OpenWhisk Serverless functions.

1. Developer *searches* and *discovers* OpenWhisk packages described by the **Package Manifest** in one or more Catalogs, that can:
 - Help analyze, augment and annotate application information and data.
 - Add value added functionality to a base application or workflow.
2. Imports Open **Package Manifest Files** and related code and artifacts into development tooling, including:
 - Project and Application (source code) Repositories
 - Integrated Development Environments (IDEs)
 - Cloud-based design, workflow and application workspaces.
3. Creates OpenWhisk **Deployment Files** for one or more target OpenWhisk enabled Clouds, with
 - Parameter values for desired target environment
 - Appropriate Credentials and configurations for chosen Event Sources and Feeds.
4. Deploys **Packages** (i.e., Actions, Triggers, Feeds, etc.) to OpenWhisk enabled Clouds, using,
 - **Package Manifest** and **Deployment File(s)**.



Notes

- Deployment Files are optional. Deployment can be fully accomplished with simply the Manifest File.

Composition

Action Sequence

An Action that is a sequenced composition of 2 or more existing Actions. The Action Sequence can be viewed as a named pipe where OpenWhisk can automatically take the output of a first Action ‘A’ in a declared sequence and provides it as input to the next Action ‘B’ in the sequence and so on until the sequence completes.

Note: This composition technique allows the reuse of existing action implementations treating them as “building blocks” for other Actions.

Namespacing

Every OpenWhisk entity (i.e., Actions, Feeds, Triggers), including packages, belongs in a *namespace*. The fully qualified name of any entity has the format:

```
/<namespaceName>[/<packageName>]/<entityName>
```


The namespace is typically provided at bind-time by the user deploying the package to their chosen OpenWhisk platform provider.

Note: The `/whisk.system` namespace is reserved for entities that are distributed with the OpenWhisk system.

Entity Names

The names of all entities, including actions, triggers, rules, packages, and namespaces, are a sequence of characters that follow the following format:

- The first character SHALL be an alphanumeric character, a digit, or an underscore.
- The subsequent characters MAY be alphanumeric, digits, spaces, or any of the following:
_, @, ., -
- The last character SHALL NOT be a space.
- The maximum name length of any entity name is 256 characters (i.e., ENTITY_NAME_MAX_LENGTH = 256).

Valid entity names are described with the following regular expression (Java metacharacter syntax):

```
"^A([\\w]|\\w|\\w@\\.){0,{ENTITY_NAME_MAX_LENGTH - 2}}[\\w@\\.])$"
```

Definitions

Activation

An invocation or “run” of an action results in an activation record that is identified by a unique activation ID. The term Activation is short-hand for the creation of this record and its information.

Repository

A location that provides storage for sets of files, as well as the history of changes made to those files.

Project

A description of a software application which enables management of its design, implementation, source control, monitoring and testing.

Application

A computer program designed to perform a group of coordinated functions, tasks, or activities to achieve some result or user benefit.

[Cloud] Service

Any resource, including a functional task, that is provided over the Internet. This includes delivery models such as *Platform as a Service* (PaaS), *Infrastructure as a Service* (IaaS), as well as *Serverless*.

Specification

This specification utilizes the [YAML language](#), a superset of JSON, which supports key features for packaging descriptors and configuration information such as built-in data types, complex data types,

anchors (relational information), files, comments and can embed other data formats such as JSON and XML easily.

YAML Types

Many of the types we use in this profile are *built-in* types from the [YAML 1.2 specification](#) (i.e., those identified by the “tag:yaml.org,2002” version tag).

The following table declares the valid YAML type URIs and aliases that SHALL be used when defining parameters or properties within an OpenWhisk package manifest:

Type Name	Type URI	Notes
string	tag:yaml.org,2002:str (default)	Default type if no type provided
integer	tag:yaml.org,2002:int	Signed. Includes large integers (i.e., long type)
float	tag:yaml.org,2002:float	Signed. Includes large floating point values (i.e., double type)
boolean	tag:yaml.org,2002:bool	This specification uses lowercase ‘true’ and lowercase ‘false’
timestamp	tag:yaml.org,2002:timestamp (see YAML-TS-1.1)	ISO 8601 compatible.
null	tag:yaml.org,2002:null	Different meaning than an empty string, map, list, etc.

Requirements

- The ‘string’ type SHALL be the default type when not specified on a parameter or property declaration.
- All ‘boolean’ values SHALL be lowercased (i.e., ‘true’ or ‘false’).

OpenWhisk Types

In addition to the YAML built-in types, OpenWhisk supports the types listed in the table below. A complete description of each of these types is provided below.

Type Name	Description	Notes
version	tag:maven.apache.org:version (see Maven version)	Typically found in modern tooling (i.e., “package@version” or “package:version” format).
string256	long length strings (e.g., descriptions)	A string type limited to 256 characters.
string64	medium length strings (e.g., abstracts, hover text)	A string type limited to 64 characters.
string16	short length strings (e.g., small form-factor list displays)	A string type limited to 16 characters.
json	The parameter value represents a JavaScript Object Notation (JSON) data object.	The deploy tool will validate the corresponding parameter value against JSON schema. Note: The implied schema for JSON is the JSON Schema (see http://json-schema.org/).

scalar-unit	Convenience type for declaring common scalars that have an associated unit. For example, “10 msec.”, “2 Gb”, etc.)	Currently, the following scalar-unit subtypes are supported: <ul style="list-style-type: none"> • scalar-unit.size • scalar-unit.time See description below for details.
schema	The parameter itself is an OpenAPI Specification v2.0 Schema Object (in YAML format) with self-defining schema.	The schema declaration follows the OpenAPI v2.0 specification for Schema Objects (YAML format).. Specifically, see https://github.com/OAI/OpenAPI-Specification/blob/master/versions/2.0.md#schemaObject
object	The parameter itself is an object with the associated defined Parameters (schemas).	Parameters of this type would include a declaration of its constituting Parameter schema.

scalar-unit types

Scalar-unit types can be used to define scalar values along with a unit from the list of recognized units (a subset of GNU units) provided below.

Grammar

```
<scalar> <unit>
```

In the above grammar, the pseudo values that appear in angle brackets have the following meaning:

- **scalar**: is a required scalar value (e.g., integer).
- **unit**: is a required unit value. The unit value MUST be type-compatible with the scalar value.

Example

```
inputs:
  max_storage_size:
    type: scalar-unit.size
    default: 10 GB
  archive_period:
    type: scalar-unit.time
    default: 30 d
```

Requirements

- Whitespace: any number of spaces (including zero or none) SHALL be allowed between the scalar value and the unit value.
- It SHALL be considered an error if either the scalar or unit portion is missing on a property or attribute declaration derived from any scalar-unit type.

Recognized units for sizes (i.e., scalar-unit.size)

Unit	Description
B	byte
kB	kilobyte (1000 bytes)

Unit	Description
MB	megabyte (1000000 bytes)
GB	gigabyte (1000000000 bytes)
TB	terabyte (1000000000000 bytes)

229 *Example*

```
inputs:
  memory_size:
    type: scalar-unit.size
    value: 256 MB
```

230 *Recognized units for times (i.e., scalar-unit.time)*

Unit	Description
d	days
h	hours
m	minutes
s	seconds
ms	milliseconds
us	microseconds

231 *Example*

```
inputs:
  max_execution_time:
    type: scalar-unit.time
    value: 600 s
```

232 *Object type example*

233 The Object type allows for complex objects to be declared as parameters with an optional
234 validateable schema.

```
inputs:
  person:
    type: object
    parameters: <schema>
```

235

236 *Schema*

237 This section defines all the essential schema used to describe OpenWhisk packages within a manifest.

238 *General Requirements*

- 239 • All field names in this specification SHALL be case sensitive.

240 *map schema*

241 The Map schema is used to define maps of key values within OpenWhisk entities.

242 *Single-line grammar*

```
{ <key_1>: <value_1>, ..., <key_n>: <value_n> }
```

243 *Multi-line grammar*

```
# Where 'key_n' is a type <string> and 'value_n' is type <any>.
<key_1>: <value_1>
...
<key_n>: <value_n>
```

244 *Examples*

245 *Single-line*

```
alert_levels: { "high": "red", "medium": "yellow", "low": green }
```

246 *Multi-line*

```
alert_levels:
  "high": "red"
  "medium": "yellow"
  "low": green
```

247

248 *Parameter schema*

249 The Parameter schema is used to define input and/or output data to be used by OpenWhisk entities for the
250 purposes of validation.

251 *Fields*

Key Name	Required	Value Type	Default	Description
type	no	<any>	string	Optional valid type name or the parameter's value for validation purposes. By default, the type is string .
description	no	string256	N/A	Optional description of the Parameter.
value	no	<any>	N/A	The optional user supplied value for the parameter. Note: this is not the default value, but an explicit declaration which allows simple usage of the Manifest file without a Deployment file..
required	no	boolean	true	Optional indicator to declare the parameter as required (i.e., true) or optional (i.e., false).
default	no	<any>	N/A	Optional default value for the optional parameters. This value MUST be type compatible with the value declared on the parameter's type field.
status	no	string	supported	Optional status of the parameter (e.g., deprecated , experimental). By default a parameter is without a declared status is considered supported.

Key Name	Required	Value Type	Default	Description
schema	no	<schema>	N/A	The optional schema if the 'type' key has the value 'schema'. The value would include a Schema Object (in YAML) as defined by the OpenAPI Specification v2.0 . This object is based upon the JSON Schema Specification .
properties	no	<list of parameter>	N/A	The optional properties if the 'type' key has the value 'object'. Its value is a listing of Parameter schema from this specification.

Requirements

- The "schema" key's value MUST be compatible with the value provided on both the "type" and "value" keys; otherwise, it is considered an error.

Notes

- The "type" key acknowledges some popular schema (e.g., JSON) to use when validating the value of the parameter. In the future additional (schema) types may be added for convenience.

Grammar

Single-line

```
<parameterName>: <YAML type> | scalar-unit | json
```

- Where <YAML type> is inferred to be a YAML type as shown in the YAML Types section above (e.g., string, integer, float, boolean, etc.).
- If you wish the parser to validate against a different schema, then the multi-line grammar MUST be used where the value would be supplied on the keyname "value" and the type (e.g., 'json') and/or schema (e.g., OpenAPI) can be supplied.

Multi-line

```
<parameterName>:
  type: <any>
  description: <string>
  required: <boolean>
  default: <any>
  status: <string>
  schema: <OpenAPI Schema Object>
```

Status values

Status Value	Description
supported (default)	Indicates the parameter is supported. This is the implied default status value for all parameters.
experimental	Indicates the parameter MAY be removed or changed in future versions.
deprecated	Indicates the parameter is no longer supported in the current version and MAY be ignored.

Dollar Notation (\$) schema for values

In a Manifest or Deployment file, a parameter value may be set from the local execution environment by using the dollar (\$) notation to denote names of local environment variables which supply the value to be inserted at execution time.

Syntax

```
<parameter>: $<local_environment_variable_name>
```

Example

```
...
inputs:
  userName: $DEFAULT_USERNAME
```

Requirements

- Processors or tooling that encounter (\$) Dollar notation and are unable to locate the value in the execution environment SHOULD resolve the value to be the default value for the type (e.g., an empty string ("") for type 'string').
- A value binding provided on the 'value' key takes precedence over a value binding on the 'default' key.

Notes

- Processors or tooling that encounter (\$) Dollar notation for values should attempt to locate the corresponding named variables set into the local execution environment (e.g., where the tool was invoked) and assign its value to the named input parameter for the OpenWhisk entity.
- This specification does not currently consider using this notation for other than simple data types (i.e., we support this mechanism for values such as strings, integers, floats, etc.) at this time.

Shared Entity Schema

The Entity Schema contains fields that are common (shared) to all OpenWhisk entities (e.g., Actions, Triggers, Rules, etc.).

Fields

Key Name	Required	Value Type	Default	Description
description	no	string256	N/A	The optional description for the Entity.
displayName	no	string16	N/A	This is the optional name that will be displayed on small form-factor devices.
annotations	no	map of <string>	N/A	The optional annotations for the Entity.

Grammar

```
description: <string256>
displayName: <string16>
annotations: <map of <string>>
```

Requirements

- Non-required fields MAY be stored as “annotations” within the OpenWhisk framework after they have been used for processing.
- Description string values SHALL be limited to 256 characters.
- DisplayName string values SHALL be limited to 16 characters.
- Annotations MAY be ignored by target consumers of the Manifest file as they are considered data non-essential to the deployment of management of OpenWhisk entities themselves.
 - Target consumers MAY preserve (persist) these values, but are not required to.
- For any OpenWhisk Entity, the maximum size of all Annotations SHALL be 256 characters.

Notes

- Several, non-normative Annotation keynames and allowed values for (principally for User Interface (UI) design) may be defined below for optional usage.

Action entity

The Action entity schema contains the necessary information to deploy an OpenWhisk function and define its deployment configurations, inputs and outputs.

Fields

Key Name	Required	Value Type	Default	Description
version	no	version	N/A	The optional user-controlled version for the Action.
function	yes	string	N/A	Required source location (path inclusive) of the Action code either <ul style="list-style-type: none">Relative to the Package manifest file.Relative to the specified Repository.
runtime	no	string	N/A	The required runtime name (and optional version) that the Action code requires for an execution environment. <i>Note: May be optional if tooling allowed to make assumptions about file extensions.</i>
inputs	no	list of parameter	N/A	The optional ordered list inputs to the Action.
outputs	no	list of parameter	N/A	The optional outputs from the Action.
limits	no	map of limit keys and values	N/A	Optional map of limit keys and their values. <i>See section “Valid limit keys” below for a listing of recognized keys and values.</i>
feed	no	boolean	false	Optional indicator that the Action supports the required parameters (and operations) to be run as a Feed Action.
web-export	no	boolean	false	Optionally, turns the Action into a “ web actions ” causing it to return HTTP content without use of an API Gateway.

Requirements

- The Action name (i.e., <actionName> MUST be less than or equal to 256 characters.

- The Action entity schema includes all general [Entity Schema](#) fields in addition to any fields declared above.
- Supplying a runtime name without a version indicates that OpenWhisk SHOULD use the most current version.
- Supplying a runtime *major version* without a *minor version* (et al.) indicates OpenWhisk SHOULD use the most current *minor version*.
- Unrecognized limit keys (and their values) SHALL be ignored.
- Invalid values for known limit keys SHALL result in an error.
- If the Feed is a Feed Action (i.e., the feed key's value is set to true), it MUST support the following parameters:
 - **lifecycleEvent**: one of 'CREATE', 'DELETE', 'PAUSE', or 'UNPAUSE'
 - These operation names MAY be supplied in lowercase (i.e., 'create', 'delete', 'pause', etc.).
 - **triggerName**: the fully-qualified name of the trigger which contains events produced from this feed.
 - **authKey**: the Basic auth. credentials of the OpenWhisk user who owns the trigger.

Notes

- Input and output parameters are implemented as JSON Objects within the OpenWhisk framework.
- The maximum code size for an Action currently must be less than 48 MB.
- The maximum payload size for an Action (i.e., POST content length or size) currently must be less than 1 MB.
- The maximum parameter size for an Action currently must be less than 1 MB.
- if no value for runtime is supplied, the value 'language:default' will be assumed.

Grammar

```
# Note: the optional [.<type>] grammar is used for naming Web Actions.
<actionName>[.<type>]:
  <Entity schema>
  version: <version>
  function: <string>
  runtime: <name>[@<[range of ]version>]
  inputs:
    <list of parameter>
  outputs:
    <list of parameter>
  limits:
    <list of limit key-values>
  feed: <boolean>
  web-export: <boolean>
```

Example

```
my_awesome_action:
  version: 1.0
  description: An awesome action written for node.js
  function: src/js/action.js
  runtime: nodejs@>0.12<6.0
  inputs:
```

```

    not_awesome_input_value:
      description: Some input string that is boring
      type: string
  outputs:
    awesome_output_value:
      description: Impressive output string
      type: string
  limits:
    memorySize: 512 kB
    logSize: 5 MB

```

Valid Runtime names

The following runtime values are currently supported by the OpenWhisk platform.

Each of these runtimes also include additional built-in packages (or libraries) that have been determined be useful for Actions surveyed and tested by the OpenWhisk platform.

These packages may vary by OpenWhisk release; examples of supported runtimes as of this specification version include:

Runtime value	OpenWhisk kind	image name	Description
nodejs	nodejs	nodejsaction:latest	Latest NodeJS runtime
nodejs@6	nodejs:6	nodejs6action:latest	Latest NodeJS 6 runtime
java, java@8	java	java8action:latest	Latest Java language runtime
python, python@2	python:2	python2action:latest	Latest Python 2 language runtime
python@3	python:3	python3action:latest	Latest Python 3 language runtime
swift, swift@2	swift	swiftaction:latest	Latest Swift 2 language runtime
swift@3	swift:3	swift3action:latest	Latest Swift 3 language runtime
swift@3.1.1	swift:3.1.1	action-swift-v3.1.1:latest	Latest Swift 3.1.1 language runtime
php	php:7.1	action-php-v7.1:latest	Latest PHP language runtime
language:default	N/A	N/A	Permit the OpenWhisk platform to select the correct default language runtime.

Recognized File extensions

Although it is best practice to provide a runtime value when declaring an Action, it is not required. In those cases, that a runtime is not provided, the package tooling will attempt to derive the correct runtime based upon the the file extension for the Action's function (source code file). The following file extensions are recognized and will be run on the latest version of corresponding Runtime listed below:

File extension	Runtime used	Description
.js	nodejs	Latest Node.js runtime.

File extension	Runtime used	Description
.java	java	Latest Java language runtime.
.py	python	Latest Python language runtime.
.swift	swift	Latest Swift language runtime.
.php	php	Latest PHP language runtime.

347 Valid Limit keys

Limit Keyname	Allowed values	Default value	Valid Range	Description
timeout	scalar-unit.time	60000 ms	[100 ms, 300000 ms]	The per-invocation Action timeout. Default unit is assumed to be milliseconds (ms).
memorySize	scalar-unit.size	256 MB	[128 MB, 512 MB]	The per-Action memory. Default unit is assumed to be in megabytes (MB).
logSize	scalar-unit.size	10 MB	[0 MB, 10 MB]	The action log size. Default unit is assumed to be in megabytes (MB).
concurrentActivations	integer	1000	See description	The maximum number of concurrent Action activations allowed (per-namespace). <i>Note: This value is not changeable via APIs at this time.</i>
userInvocationRate	integer	5000	See description	The maximum number of Action invocations allowed per user, per minute. <i>Note: This value is not changeable via APIs at this time.</i>
codeSize	scalar-unit.size	48 MB	See description	The maximum size of the Action code. <i>Note: This value is not changeable via APIs at this time.</i>
parameterSize	scalar-unit.size	1 MB	See description	The maximum size <i>Note: This value is not changeable via APIs at this time.</i>

348 Notes

349 The default values and ranges for limit configurations reflect the defaults for the OpenWhisk platform
350 (open source code). These values may be changed over time to reflect the open source community
351 consensus.

352 Web Actions

353 OpenWhisk can turn any Action into a “web action” causing it to return HTTP content without use of an
354 API Gateway. Simply supply a supported “type” extension to indicate which content type is to be
355 returned and identified in the HTTP header (e.g., .json, .html, .text or .http).

Return values from the Action's function are used to construct the HTTP response. The following response parameters are supported in the response object.

- **headers:** a JSON object where the keys are header-names and the values are string values for those headers (default is no headers).
- **code:** a valid HTTP status code (default is 200 OK).
- **body:** a string which is either plain text or a base64 encoded string (for binary data).

Trigger entity

The Trigger entity schema contains the necessary information to describe the stream of events that it represents. For more information, see the document “[Creating Triggers and Rules](#)”.

Fields

Key Name	Required	Value Type	Default	Description
feed	no	string	N/A	The optional name of the Feed associated with the Trigger.
credential	no	Credential	N/A	The optional credential used to access the feed service.
inputs	no	list of parameter	N/A	The optional ordered list inputs to the feed.
events	no	list of Event	N/A	The optional list of valid Event schema the trigger supports. OpenWhisk would validate incoming Event data for conformance against any Event schema declared under this key. <i>Note: This feature is <u>not supported at this time</u>. This is viewed as a possible feature that may be implemented along with configurable options for handling of invalid events.</i>

Requirements

- The Trigger name (i.e., <triggerName> MUST be less than or equal to 256 characters.
- The Trigger entity schema includes all general [Entity Schema](#) fields in addition to any fields declared above.

Notes

- The 'events' key name is not supported at this time.
- The Trigger entity within the OpenWhisk programming model is considered outside the scope of the Package (although there are discussions about changing this in the future). This means that Trigger and API information will not be returned when using the OpenWhisk Package API:
 - `wsk package list <package name>`
- However, it may be obtained using the Trigger API:
 - `wsk trigger list -v`

Grammar

```
<triggerName>:  
  <Entity schema>  
  feed: <feed name>  
  credential: <Credential>
```

```
inputs:
  <list of parameter>
```

379 *Example*

```
triggers:
  everyhour:
    feed: /whisk.system/alarms/alarm
```

380 *Rule entity*

381 The Rule entity schema contains the information necessary to associates one trigger with one action, with
382 every firing of the trigger causing the corresponding action to be invoked with the trigger event as input.
383 For more information see the document “[Creating Triggers and Rules](#)”.

384 *Fields*

Key Name	Required	Value Type	Default	Description
trigger	yes	string	N/A	Required name of the Trigger the Rule applies to.
action	yes	string	N/A	Required name of the Action the Rule applies to.
rule	no	regex	true	The optional regular expression that determines if the Action is fired. Note: In this version of the specification, only the expression “true” is currently supported.

385 *Requirements*

- 386 • The Rule name (i.e., <ruleName>) MUST be less than or equal to 256 characters.
- 387 • The Rule entity schema includes all general [Entity Schema](#) fields in addition to any fields declared
- 388 above.

389 *Requirements*

- 390 • OpenWhisk only supports a value of 'true' for the 'rule' key's value at this time.

391 *Grammar*

```
<ruleName>:
  description: <string>
  trigger: <string>
  action: <string>
  rule: <regex>
```

392 *Example*

```
my_rule:
  description: Enable events for my Action
  trigger: my_trigger
  action: my_action
```

Feed entity

The OpenWhisk Feed entity schema contains the information necessary to describe a configurable service (that may work with an existing network accessible service) to produce events on its behalf thereby acting as an Event Source.

At this time, the Package Manifest simply provides the information to describe a Feed (service), its Action, lifecycle operations (along with their parameters) and the associated service it works with. In the future, we intend to allow more granular ability to manage Feeds directly using their operations.

Fields

Key Name	Required	Value Type	Default	Description
location	no	string	N/A	The URL for the Feed service which can be used by the OpenWhisk platform for registration and configuration.
credential	no	string	N/A	Contains either: <ul style="list-style-type: none">• A credential string.• The optional name of a credential (e.g., token) that must be used to access the Feed service. Note: this would be defined elsewhere, perhaps as an input parameter to the Package.
operations	no	list of operations	N/A	The list of operations (i.e., APIs) the Feed supports on the URL provided described, by default, using the OpenAPI (f.k.a. "Swagger") specification schema .
operation_type	no	openwhisk openapi@<version>	openwhisk	The specification format for the operation definitions.
action	no	string	N/A	The optional name of the Action if this is a Feed Action, that is, the Feed service implementation is an OpenWhisk Action.

Requirements

- The Feed name (i.e., <feedName> MUST be less than or equal to 256 characters.
- The Feed entity schema includes all general [Entity Schema](#) fields in addition to any fields declared above.
- If the action field is set, the corresponding Action definition and function (code) MUST be a valid Feed Action.
- The location and credential SHOULD be supplied if the Feed is not a Feed action using a Deployment File.
- Operation names in manifests MAY be lower or upper cased (e.g., "create" or "CREATE").

Grammar

```
<feedName>:  
  location: <string>  
  credential: <string>  
  operations:  
    <list of operations>
```

```
action: <string>
```

Example

The following example shows the mandatory operations for Feed Actions.

```
my_feed:
  description: A simple event feed
  location: https://my.company.com/services/eventHub
  # Reference to a credential defined elsewhere in manifest
  credential: my_credential
  operations:
    # Note: operation names in manifests MAY be lower or upper cased.
    create | CREATE:
      inputs:
        <parameters>
    delete | DELETE:
      inputs:
        <parameters>
    pause | PAUSE:
      inputs:
        <parameters>
    unpause | UNPAUSE:
      inputs:
        <parameters>
    # Additional, optional operations
    ...
```

Discussion

For a description of types of Feeds and why they exist, please see:

- <https://github.com/apache/incubator-openwhisk/blob/master/docs/feeds.md>.

Feed Actions

OpenWhisk supports an open API, where any user can expose an event producer service as a **feed** in a **package**. This section describes architectural and implementation options for providing your own feed.

Feed actions and Lifecycle Operations

The *feed action* is a normal OpenWhisk *action*, but it should accept the following parameters:

- **lifecycleEvent**: one of 'CREATE', 'DELETE', 'PAUSE', or 'UNPAUSE'
- **triggerName**: the fully-qualified name of the trigger which contains events produced from this feed.
- **authKey**: the Basic auth. credentials of the OpenWhisk user who owns the trigger just mentioned

The feed action can also accept any other parameters it needs to manage the feed. For example, the Cloudant changes feed action expects to receive parameters including 'dbname', 'username', etc.

Sequence entity

Actions can be composed into sequences to, in effect, form a new Action. The Sequence entity allows for a simple, convenient way to describe them in the Package Manifest.

431 *Fields*

Key Name	Required	Value Type	Default	Description
actions	yes	list of Action	N/A	• The required list of two or more actions

432 *Requirements*

- 433 • The comma separated list of Actions on the actions key SHALL imply the order of the sequence (from
434 left, to right).
- 435 • There MUST be two (2) or more actions declared in the sequence.

436 *Notes*

- 437 • The sequences key exists for convenience; however, it is just one possible instance of a composition
438 of Actions. The composition entity is provided for not only describing sequences, but also for other
439 (future) compositions and additional information needed to compose them. For example, the
440 composition entity allows for more complex mappings of input and output parameters between
441 Actions.

442 *Grammar*

```
sequences:  
  <sequence name>:  
    <Entity schema>  
    actions: <ordered list of action names>  
    ...
```

443 *Example*

```
sequences:  
  newbot:  
    actions: oauth/login, newbot-setup, newbot-greeting
```

444 *API entity*

445 This entity allows manifests to link Actions to be made available as HTTP-based API endpoints as
446 supported by the API Gateway service of OpenWhisk.

447 This entity declaration is intended to provide grammar for the experimental API (*see*
448 <https://github.com/apache/incubator-openwhisk/blob/master/docs/apigateway.md> and shown using a
449 "book club" example:

450 *CLI Example*

```
$ wsk api create -n "Book Club" /club /books get getBooks  
$ wsk api create /club /books post postBooks  
$ wsk api create /club /books put putBooks  
$ wsk api create /club /books delete deleteBooks
```

451 the above would translate to the following grammars in the pkg. spec. to a new-top level keyname "apis"
452 in the manifest:

453 Grammar

```
apis:
  <API name>:          # descriptive name
    description: <string> # optional, description
    <basepath>:         # shared basepath
      <path>:
        <action name>: get | post | put | delete
        ...
      ...
```

454 Note

- 455 • There can be more than one set of named <path> actions under the same <basepath>.

456 Example

457 A somewhat simplified grammar is also supported that allows single-line definition of Actions (names)
458 along with their HTTP verbs.

```
459 apis:
    book-club:
      club:
        books:
          getBooks: get
          postBooks: post
          putBooks: put
          deleteBooks: delete
        members:
          listMembers: get
```

460 Requirements

- 461 • The API entity's name (i.e., <API Name>) MUST be less than or equal to 256 characters.

462 Notes

- 463 • The API entity within the OpenWhisk programming model is considered outside the scope of the
464 Package. This means that API information will not be returned when using the OpenWhisk Package
465 API:
 - 466 • `wsk package list <package name>`
- 467 • However, it may be obtained using the Trigger API:
 - 468 • `wsk api list -v`

469 Package entity

470 The Package entity schema is used to define an OpenWhisk package within a manifest.

471 Fields

Key Name	Required	Value Type	Default	Description
version	yes	version	N/A	The required user-controlled version for the Package.

Key Name	Required	Value Type	Default	Description
license	no	string	N/A	The required value that indicates the type of license the Package is governed by. The value is required to be a valid Linux-SPDX value. See https://spdx.org/licenses/ .
credential	no	string	N/A	The optional Credential used for all entities within the Package. The value is either: Contains either: <ul style="list-style-type: none"> • A credential string. • The optional name of a credential (e.g., token) that is defined elsewhere.
dependencies	no	list of Dependency	N/A	The optional list of external OpenWhisk packages the manifest needs deployed before it can be deployed.
repositories	no	list of Repository	N/A	The optional list of external repositories that contain functions and other artifacts that can be found by tooling.
actions	no	list of Action	N/A	Optional list of OpenWhisk Action entity definitions.
sequences	no	list of Sequence	N/A	Optional list of OpenWhisk Sequence entity definitions.
triggers	no	list of Trigger	N/A	Optional list of OpenWhisk Trigger entity definitions.
rules	no	list of Rule	N/A	Optional list of OpenWhisk Rule entity definitions.
feeds	no	list of Feed	N/A	Optional list of OpenWhisk Feed entity definitions.
apis	no	list of API	N/A	Optional list of API entity definitions.
compositions (Not yet supported)	no	list of Composition	N/A	Optional list of OpenWhisk Composition entity definitions.

Requirements

- The Package name MUST be less than or equal to 256 characters.
- The Package entity schema includes all general [Entity Schema](#) fields in addition to any fields declared above.
- A valid Package license value MUST be one of the [Linux SPDX](#) license values; for example: Apache-2.0 or GPL-2.0+, or the value 'unlicensed'.
- Multiple (mixed) licenses MAY be described using using [NPM SPDX license syntax](#).
- A valid Package entity MUST have one or more valid Actions defined.

Notes

- Currently, the 'version' value is not stored in Apache OpenWhisk, but there are plans to support it in the future.
- Currently, the 'license' value is not stored in Apache OpenWhisk, but there are plans to support it in the future.

- 485 • The Trigger and API entities within the OpenWhisk programming model are considered outside the
486 scope of the Package. This means that Trigger and API information will not be returned when using
487 the OpenWhisk Package API:
- 488 • `wsk package list <package name>`
- 489 • However, their information may be retrieved using respectively:
- 490 • `wsk trigger list -v`
- 491 • `wsk api list -v`

492 Grammar

```
<packageName>:
  <Entity schema>
  version: <version>
  license: <string>
  repositories: <list of Repository>
  actions: <list of Action>
  sequences: <list of Sequence>
  triggers: <list of Trigger>
  rules: <list of Rule>
  feeds: <list of Feed>
  apis: <list of API>
  compositions: <list of Composition> # Not yet supported
```

493 Example

```
my_whisk_package:
  description: A complete package for my awesome action to be deployed
  version: 1.2.0
  license: Apache-2.0
  actions:
    my_awesome_action:
      <Action schema>
  triggers:
    trigger_for_awesome_action:
      <Trigger schema>
  rules:
    rule_for_awesome_action:
      <Rule schema>
```

494 Composition entity (Not yet supported)

495 The Composition entity schema contains information to declare compositions of OpenWhisk Actions.
496 Currently, this includes Action Sequences where Actions can be composed of two or more existing
497 Actions.

498 Fields

Key Name	Required	Value Type	Default	Description
type	no	string	sequence	The optional type of Action composition. <i>Note: currently only 'sequence' is supported.</i>

Key Name	Required	Value Type	Default	Description
inputs	no	list of parameter	N/A	The optional list of parameters for the Action composition (e.g., Action Sequence).
outputs	no	list of parameter	N/A	The optional outputs from the Entity.
sequence	no	ordered list of Action (names)	N/A	The optional expression that describes the connections between the Actions that comprise the Action sequence composition.
parameterMappings	no	TBD	N/A	<p>The optional expression that describes the mappings of parameter (names and values) between the outputs of one Action to the inputs of another Action.</p> <p>Note: Currently, mappings are not supported and JSON objects are passed between each Action in a sequence. At this time, it is assumed that the Actions in a sequence are designed to work together with no output to input mappings being performed by the OpenWhisk platform.</p>

499 *Requirements*

- 500 • The Composition name (i.e., <compositionName> MUST be less than or equal to 256 characters.
- 501 • The Composition entity schema includes all general [Entity Schema](#) fields in addition to any fields
- 502 declared above.

503 *Grammar*

```

<compositionName>:
  <Entity schema> # Common to all OpenWhisk Entities
  type: <string>
  inputs:
    <list of parameter>
  outputs:
    <list of parameter>
  sequence:
    actions: <ordered list of action names>
  parameterMappings:
    # TBD. This is a future use case.

```

504 *Example: multi-line sequence*

```

my_action_sequence:
  type: sequence
  sequence:
    actions: action_1, action_2, action_3
  inputs:
    simple_input_string: string
  outputs:
    annotated_output_string: string

```

Extended Schema

Dependencies

The dependencies section allows you to declare other OpenWhisk packages that your application or project (manifest) are dependent on. A Dependency is used to declare these other packages which deployment tools can use to automate installation of these pre-requisites.

Fields

Key Name	Required	Value Type	Default	Description
location	yes	string	N/A	The required location of the dependent package.
version	yes	version	N/A	The required version of the dependent package.
inputs	no	list of parameter	N/A	The optional Inputs to the dependent package.

Requirements

- No additional requirements.

Notes

- The <package_name> is a local alias for the actual package name as described in the referenced package. The referenced package would have its own Manifest file that would include its actual Package name (and the one that would be used by the wskdeploy tool to replace the local alias).
- The 'version' parameter is currently used to specify a branch in GitHub and defaults to "master", this behavior may change in upcoming releases of the specification.
- The experimental key name 'name' is only valid when the deprecated 'package' keyword has been used instead of the favored key 'packages'. If it is used within the 'packages' structure, it will cause a warning and be ignored as it is redundant to the <packageName>.

Grammar

```
<package_name>:  
  <Entity schema>  
  location: <GitHub URL> |  
  version: 1.0.1  
  inputs:  
    <list of parameter>
```

Example

```
dependencies:  
  status_update:  
    location: github.com/myrepo/statusupdate  
    version: 1.0  
  database_pkg:  
    location: /whisk.system/couchdb  
    inputs:  
      dbname: MyAppDB
```

Repository

A repository defines a named external repository which contains (Action) code or other artifacts package processors can access during deployment.

Fields

Key Name	Required	Value Type	Default	Description
description	no	string256	N/A	Optional description for the Repository.
url	yes	string	N/A	Required URL for the Repository.
credential	no	Credential	N/A	Optional name of a Credential defined in the Package that can be used to access the Repository.

Requirements

- The Repository name (i.e., <repositoryName> MUST be less than or equal to 256 characters.
- Description string values SHALL be limited to 256 characters.

Grammar

Single-line (no credential)

```
<repositoryName>: <repository_address>
```

Multi-line

```
<repositoryName>:  
  description: <string256>  
  url: <string>  
  credential: <Credential>
```

Example

```
my_code_repo:  
  description: My project's code repository in GitHub  
  url: https://github.com/openwhisk/openwhisk-package-rss
```

Credential

A Credential is used to define credentials used to access network accessible resources. Fields

Key Name	Required	Value Type	Default	Description
protocol	no	string	N/A	Optional protocol name used to indicate the authorization protocol to be used with the Credential's token and other values.
tokenType	yes	string	password	Required token type used to indicate the type (format) of the token string within the supported types allowed by the protocol.

Key Name	Required	Value Type	Default	Description
token	yes	string	N/A	Required token used as a credential for authorization or access to a networked resource.
description	no	string256	N/A	Optional description for the Credential.
keys	no	map of string	N/A	Optional list of protocol-specific keys or assertions.

Requirements

- The Credential name (i.e., <credentialName> MUST be less than or equal to 256 characters.
- Description string values SHALL be limited to 256 characters.

Valid protocol values

Protocol Value	Valid Token Type Values	Description
plain	N/A	Basic (plain text) username-password (no standard).
http	basic_auth	HTTP Basic Authentication Protocol.
xauth	X-Auth-Token	HTTP Extended Authentication Protocol (base-64 encoded Tokens).
oauth	bearer	Oauth 2.0 Protocol
ssh	identifier	SSH Keypair protocol (e.g., as used in OpenStack)

Grammar

```
Credential:
  type: Object
  properties:
    protocol:
      type: string
      required: false
    tokenType:
      type: string
      default: password
    token:
      type: string
    keys:
      type: map
      required: false
    entry_schema:
      type: string
    user:
      type: string
      required: false
```

Notes

- The use of transparent user names (IDs) or passwords are not considered best practice.

549 *Examples*

550 *Plain username-password (no standardized protocol)*

```
inputs:
  my_credential:
    type: Credential
    properties:
      user: my_username
      token: my_password
```

551 *HTTP Basic access authentication*

```
inputs:
  my_credential:
    type: Credential
    description: Basic auth. where <username>:<password> are a single string
    properties:
      protocol: http
      token_type: basic_auth
      # Note: this would be base64 encoded before transmission by any impl.
      token: myusername:mypassword
```

552 *X-Auth-Token*

```
inputs:
  my_credential:
    type: Credential
    description: X-Auth-Token, encoded in Base64
    properties:
      protocol: xauth
      token_type: X-Auth-Token
      # token encoded in Base64
      token: 604bbe45ac7143a79e14f3158df67091
```

553 *OAuth bearer token*

```
inputs:
  my_credential:
    type: Credential
    properties:
      protocol: oauth2
      token_type: bearer
      # token encoded in Base64
      token: 8ao9nE2DEjr1zCsicWMpBC
```

554 *SSH Keypair*

```
inputs:
  my_ssh_keypair:
    type: Credential
    properties:
      protocol: ssh
      token_type: identifier
      # token is a reference (ID) to an existing keypair (already installed)
```



```
token: <keypair_id>
```

Package Artifacts

Package Manifest File

The Package Manifest file is the primary OpenWhisk Entity used to describe an OpenWhisk Package and all necessary **schema** and **file** information needed for deployment. It contains the [Package entity schema](#) described above.

Deployment File

The Deployment file is used in conjunction with a corresponding Package Manifest file to provide configuration information (e.g., input parameters, authorization credentials, etc.) needed to deploy, configure and run an OpenWhisk Package for a target Cloud environment.

The manifest and Deployment files are comprised of the following entities:

Application

An optional, top-level key that describes a set of related Packages that together comprise a higher-order application.

Fields

Key Name	Required	Value Type	Default	Description
version	no	version	N/A	The optional user-controlled version for the Application.
name	no	string256	N/A	The optional name of the application. Note: This key is only valid in the singular 'package' grammar.
namespace	no	string	N/A	The optional namespace for the application (and default namespace for its packages where not specified).
credential	no	string	N/A	The optional credential for the application (and default credential for its packages where not specified).
package	maybe	package (singular)	N/A	The required package definition when the key name 'packages' (plural) is not present.
packages	maybe	list of package (plural)	N/A	The required list of <u>one or more</u> package definitions when the key name 'package' (singular) is not present.

Grammar (singular)

```
application:  
  version: <version>
```

```
name: <string256>
namespace: <string>
credential: <string>
package:
  <package definition>
```

574 *Grammar (plural)*

```
application:
  version: <version>
  name: <string256>
  namespace: <string>
  credential: <string>
  packages:
    <list of package definitions>
```

575 *Requirements*

- 576 • The keys under the application key (e.g., name, namespace, credential and packages) are only used in
577 a manifest or deployment file if the optional application key is used.
- 578 • Either the key name 'package' (singular) or the key name 'packages' (plural) MUST be provided but
579 not both.
 - 580 ○ If the 'package' key name is provided, its value must be a valid package definition.
 - 581 ○ If the 'packages' key name is provided, its value must be one or more valid package
582 definitions.

583 *Notes*

- 584 • Currently, the OpenWhisk platform does not recognize the Application entity as part of the
585 programming model; it exists as a higher order grouping concept only in this specification. Therefore,
586 there is no data stored within OpenWhisk for the Application entity.
- 587 • The keyword 'package' and its singular grammar for declaring packages MAY be deprecated in future
588 versions of the specification.

589 *Example*

```
application:
  name: greetings
  namespace: /mycompany/greetings/
  credential: 1234-5678-90abcdef-0000
  packages:
    helloworld:
      inputs:
        city: Boston
      actions:
        hello:
          inputs: # input bindings
            personName: Paul
  ...
```

590 *Example Notes*

- 591 • A common use would be to associate a namespace (i.e., a target namespace binding) or credential to
592 an application and all included packages automatically inherit that namespace (if applied at that
593 level) unless otherwise provided (similar to style inheritance in CSS).
- 594 • The application name would be treated as metadata, perhaps stored in the annotations for the
595 contained entities.

Normative References

Tag	Description
RFC2119	S. Bradner, <i>Key words for use in RFCs to Indicate Requirement Levels</i> , http://www.ietf.org/rfc/rfc2119.txt , IETF RFC 2119, March 1997.
YAML-1.2	YAML, Version 1.2, 3rd Edition, Patched at 2009-10-01, Oren Ben-Kiki, Clark Evans, Ingy döt Net http://www.yaml.org/spec/1.2/spec.html
YAML-TS-1.1	Timestamp Language-Independent Type for YAML Version 1.1, Working Draft 2005-01-18, http://yaml.org/type/timestamp.html
Maven-Version	The version type is defined with the Apache Maven project's policy draft: https://cwiki.apache.org/confluence/display/MAVEN/Version+number+policy
OpenAPI-2.0	The OpenAPI (f.k.a. "Swagger") specification for defining REST APIs as JSON. https://github.com/OAI/OpenAPI-Specification/blob/master/versions/2.0.md
Linux-SPDX	Linux Foundation, SPDX License list https://spdx.org/licenses/
NPM-SPDX-Syntax	Node Package Manager (NPM) SPDX License Expression Syntax https://www.npmjs.com/package/spdx

Non-normative References

Tag	Description
OpenWhisk-API	OpenWhisk REST API which is defined as an OpenAPI document. https://raw.githubusercontent.com/openwhisk/openwhisk/master/core/controller/src/main/resources/whiskswagger.json
GNU-units	Size-type units are based upon a subset of those defined by GNU at http://www.gnu.org/software/parted/manual/html_node/unit.html
RFC 6838	Mime Type definitions in compliance with RFC 6838 .
RFC 7231	HTTP 1.1. status codes are described in compliance with RFC 7231 .
IANA-Status-Codes	HTTP Status codes as defined in the IANA Status Code Registry .
JSON Schema Specification	The built-in parameter type "json" references this specification. http://json-schema.org/

Scenarios and Use cases

Usage Scenarios

User background

The following assumptions about the users referenced in the usage scenarios:

- Experienced developer; knows Java, Node, SQL, REST principles and basic DevOps processes; uses IDEs to develop code locally.
- Limited exposure to Serverless, but interested in trying new technologies that might improve productivity.

Scenario 1: Clone and Create

Deploy an OpenWhisk app (project, set of entities, package, ...) discovered on github. The developer...

1. discovers an interesting git repo containing an OpenWhisk app (project, set of entities, package, ...)
2. clones the repo to local disk.
3. He pushes (deploys) it into one of his OpenWhisk namespaces
4. He checks out the app's behavior using OpenWhisk CLI or OpenWhisk UI

Notes

- while this scenario allows to use the manifest file as a "black box" the manifest format can influence the user experience of a developer trying to read it and understand what it does

Scenario 2: Pushing Updates with versioning

Change a cloned repo that he previously pushed into one of his namespaces. The developer...

1. changes the local repo by editing code and adding and changing entity specifications using local tools (editors, IDEs, ...).
2. bumps version number for package.
3. pushes his updates into the namespace so that the existing entities are changed accordingly.

Scenario 3: Start New Repo with Manifest

Start a new OpenWhisk app (project, set of entities) from scratch. The developer...

1. code files for the actions (e.g. *action1.js*, *action2.js*, *action3.js*)
2. creates a *LICENSE.txt* file
3. Creates a **Manifest File** that specifies the set of OpenWhisk entities and their relations (e.g. *manifest.yml*). It also references the *LICENSE.txt* file.
4. initializes and uploads the set of files as a new git repo.

634 *Notes:*

- 635 • Creating the initial manifest file should be supported by providing an empty template with syntax
636 examples and other helpful comments

637 *Scenario 4: Export into Repository*

638 Share an existing OpenWhisk app (project, set of entities) with others
639 so that they can deploy and change it for their purposes. The developer...

- 640 1. exports a defined set of entities (a whole namespace?) into a set of files that includes code files,
641 and generated manifest, LICENSE.txt and README files.
- 642 2. initializes and uploads the set of files as a new git repo.
643 Example: `git init` ... etc.

644 *Scenario 5: Discovery and Import from object store*

645 Discover an OpenWhisk package (manifest) co-located with data in an Object storage service.

646 This package would include a description of the Actions, Triggers, Rules and Event Sources (or Feeds)
647 necessary to interact with data it is associated with directly from the Object storage repository; thus
648 allowing anyone with access to the data an immediate way to interact and use the data via the OpenWhisk
649 Serverless platform.

650

Guided examples

This packaging specification grammar places an emphasis on simplicity for the casual developer who may wish to hand-code a Manifest File; however, it also provides a robust optional schema that can be advantaged when integrating with larger application projects using design and development tooling such as IDEs.

This guide will use examples to incrementally show how to use the OpenWhisk Packaging Specification to author increasingly more interesting Package Manifest and Deployment files taking full advantage of the specification's schema.

Please note that the Apache 'wskdeploy' utility will be used to demonstrate output results.

Package Examples

Example 1: Minimal valid Package Manifest

This use case shows a minimal valid package manifest file.

including:

- shows how to declare a Package named 'hello_world_package'.

Manifest Files

Example 1: Minimum valid Package manifest file

```
package:
  name: hello_world_package
  version: 1.0
  license: Apache-2.0
```

Notes

- Currently, the 'version' and 'license' key values are not stored in Apache OpenWhisk, but there are plans to support it in the future.

Actions Examples

Example 1: The "Hello world" Action

As with most language introductions, in this first example we encode a simple "hello world" action, written in JavaScript, using an OpenWhisk Package Manifest YAML file.

It shows how to:

- declare a single Action named 'hello_world' within the 'hello_world_package' Package.
- associate the JavaScript function's source code, stored in the file 'src/hello.js', to the 'hello_world' Action.

682 **Manifest File**

683 *Example: "Hello world" using a NodeJS (JavaScript) action*

```
package:
  name: hello_world_package
  version: 1.0
  license: Apache-2.0
  actions:
    hello_world:
      function: src/hello.js
```

684

685 where "hello.js", within the package-relative subdirectory named 'src', contains the following
686 JavaScript code:

```
function main(params) {
  msg = "Hello, " + params.name + " from " + params.place;
  return { greeting: msg };
}
```

687 **Deploying**

```
$ ./wskdeploy -m docs/examples/manifest_hello_world.yaml
```

688 **Invoking**

```
$ wsk action invoke hello_world_package/hello_world --blocking
```

689 **Result**

690 The invocation should return an 'ok' with a response that includes this result:

```
"result": {
  "greeting": "Hello, undefined from undefined"
},
```

691 The output parameter 'greeting' contains "*undefined*" values for the 'name' and 'place' input
692 parameters as they were not provided in the manifest.

693 **Discussion**

694 This "hello world" example represents the minimum valid Manifest file which includes only the required
695 parts of the Package and Action descriptors.

696
697 In the above example,

- 698 • The Package and its Action were deployed to the user's default namespace using the 'package' name.
699 • `/<default namespace>/hello_world_package/hello_world`
- 700 • The NodeJS default runtime (i.e., `runtime: nodejs`) was automatically selected based upon the '.js'
701 extension on the Action function's source file 'hello.js'.

Example 2: Adding fixed Input values to an Action

This example builds upon the [previous “hello world” example](#) and shows how fixed values can be supplied to the input parameters of an Action.

It shows how to:

- declare input parameters on the action ‘hello_world’ using a single-line grammar.
- add ‘name’ and ‘place’ as input parameters with the fixed values “Sam” and “the Shire” respectively.

Manifest File

Example: “Hello world” with fixed input values for ‘name’ and ‘place’

```
package:
  name: hello_world_package
  version: 1.0
  license: Apache-2.0
  actions:
    hello_world_fixed_parms:
      function: src/hello.js
      inputs:
        name: Sam
        place: the Shire
```

Deployment

```
$ ./wskdeploy -m docs/examples/manifest_hello_world_fixed_parms.yaml
```

Invoking

```
$ wsk action invoke hello_world_package/hello_world_fixed_parms --blocking
```

Result

The invocation should return an 'ok' with a response that includes this result:

```
"result": {
  "greeting": "Hello, Sam from the Shire"
},
```

Discussion

In this example:

- The value for the ‘name’ input parameter would be set to “Sam”.
- The value for the ‘place’ input parameter would be set to “the Shire”.
- The wskdeploy utility would infer that both ‘name’ and ‘place’ input parameters to be of type ‘string’.

Example 3: “Hello world” with typed input and output parameters

This example shows the “Hello world” example with typed input and output Parameters.

723
724 It shows how to:

- 725 • declare input and output parameters on the action 'hello_world' using a simple, single-line
- 726 grammar.
- 727 • add two input parameters, 'name' and 'place', both of type 'string' to the 'hello_world' action.
- 728 • add an 'integer' parameter, 'age', to the action.
- 729 • add a 'float' parameter, 'height', to the action.
- 730 • add two output parameters, 'greeting' and 'details', both of type 'string', to the action.

731 **Manifest File**

732 *Example: "Hello world" with typed input and output parameter declarations*

```
package:
  name: hello_world_package
  ... # Package keys omitted for brevity
  actions:
    hello_world_typed_parms:
      function: src/hello_plus.js
      inputs:
        name: string
        place: string
        children: integer
        height: float
      outputs:
        greeting: string
        details: string
```

733 where the function 'hello_plus.js', within the package-relative subdirectory named 'src', is
734 updated to use the new parameters:

```
function main(params) {
  msg = "Hello, " + params.name + " from " + params.place;
  family = "You have " + params.children + " children ";
  stats = "and are " + params.height + " m. tall.";
  return { greeting: msg, details: family + stats };
}
```

735 **Deployment**

```
$ ./wskdeploy -m docs/examples/manifest_hello_world_typed_parms.yaml
```

736 **Invoking**

```
$ wsk action invoke hello_world_package/hello_world_typed_parms --blocking
```

737 **Result**

738 The invocation should return an 'ok' with a response that includes this result:

```
"result": {
  "details": "You have 0 children and are 0 m. tall.",
```

```
"greeting": "Hello,  from "
},
```

Discussion

In this example:

- The default value for the 'string' type is the empty string (i.e., ""); it was assigned to the 'name' and 'place' input parameters.
- The default value for the 'integer' type is zero (0); it was assigned to the 'age' input parameter.
- The default value for the 'float' type is zero (0.0f); it was assigned to the 'height' input parameter.

Example 4: “Hello world” with advanced parameters

This example builds on the previous “[Hello world” with typed input and output parameters](#)’ example with more robust input and output parameter declarations by using a multi-line format for declaration.

This example:

- shows how to declare input and output parameters on the action ‘hello_world’ using a multi-line grammar.

Manifest file

If we want to do more than declare the type (i.e., ‘string’, ‘integer’, ‘float’, etc.) of the input parameter, we can use specifications the multi-line grammar for Parameters.

Example: input and output parameters with advanced fields

```
package:
  name: hello_world_package
  ... # Package keys omitted for brevity
  actions:
    hello_world_advanced_parms:
      function: src/hello.js
      inputs:
        name:
          type: string
          description: name of person
          default: unknown person
        place:
          type: string
          description: location of person
          value: the Shire
        children:
          type: integer
          description: Number of children
          default: 0
        height:
          type: float
          description: height in meters
          default: 0.0
      outputs:
        greeting:
```

```
    type: string
    description: greeting string
  details:
    type: string
    description: detailed information about the person
```

756 **Deployment**

```
$ ./wskdeploy -m docs/examples/manifest_hello_world_advanced_parms.yaml
```

757 **Invoking**

```
$ wsk action invoke hello_world_package/hello_world_advanced_parms --blocking
```

758 Invoking the action would result in the following response:

```
"result":
  "details": "You have 0 children and are 0 m. tall.",
  "greeting": "Hello, unknown person from the Shire"
},
```

759 **Discussion**

- 760 • Describing the input and output parameter types, descriptions, defaults and other data:
 - 761 ○ enables tooling to validate values users may input and prompt for missing values using the
 - 762 descriptions provided.
 - 763 ○ allows verification that outputs of an Action are compatible with the expected inputs of another
 - 764 Action so that they can be composed in a sequence.
- 765 • The 'name' input parameter was assigned the 'default' key's value "unknown person".
- 766 • The 'place' input parameter was assigned the 'value' key's value "the Shire".

767 **Example 5: Adding a Trigger and Rule to “hello world”**

768 This example will demonstrate how to define a Trigger that is compatible with the basic ‘hello_world’
769 Action and associate it using a Rule.

770 **Manifest File**

771 *Example: “Hello world” Action with a compatible Trigger and Rule*

```
package:
  name: hello_world_package
  ... # Package keys omitted for brevity
actions:
  hello_world_triggerrule:
    function: src/hello_plus.js
    inputs:
      name: string
      place: string
      children: integer
      height: float
```

```

    outputs:
      greeting: string
      details: string

    triggers:
      meetPerson:
        inputs:
          name: Sam
          place: the Shire
          children: 13
          height: 1.2

    rules:
      myPersonRule:
        trigger: meetPerson
        action: hello_world_triggerrule

```

772 **Deployment**

773 *without the Deployment file:*

```
$ wskdeploy -m docs/examples/manifest_hello_world_triggerrule.yaml
```

774 **Invoking**

775 First, let's try *"invoking"* the 'hello_world_triggerrule' Action directly without the Trigger.

```
$ wsk action invoke hello_world_package/hello_world_triggerrule --blocking
```

776 Invoking the action would result in the following response:

```

"result": {
  "details": "You have 0 children and are 0 m. tall.",
  "greeting": "Hello, from "
},

```

777 As you can see, the results verify that the default values (i.e., empty strings and zeros) for the input
 778 parameters on the 'hello_world_triggerrule' Action were used to compose the 'greeting' and
 779 'details' output parameters. This is as expected since we did not bind any values or provide any
 780 defaults when we defined the 'hello_world_triggerrule' Action in the manifest file.

781 **Triggering**

782 Instead of invoking the Action, here try *"firing"* the 'meetPerson' Trigger:

```
$ wsk trigger fire meetPerson
```

783 **Result**

784 which results in an Activation ID:

```
ok: triggered /_/meetPerson with id a8e9246777a7499b85c4790280318404
```

785 The 'meetPerson' Trigger is associated with 'hello_world_triggerrule' Action the via the
786 'meetPersonRule' Rule. We can verify that firing the Trigger indeed cause the Rule to be activated
787 which in turn casues the Action to be invoked:

```
$ wsk activation list

d03ee729428d4f31bd7f61d8d3ecc043 hello_world_triggerrule
3e10a54cb6914b37a8abca53596dcc9 meetPersonRule
5ff4804336254bfba045ceaa1eeb4182 meetPerson
```

788 we can then use the 'hello_world_triggerrule' Action's Activation ID to see the result:

```
$ wsk activation get d03ee729428d4f31bd7f61d8d3ecc043
```

789 to view the actual results from the action:

```
"result": {
  "details": "You have 13 children and are 1.2 m. tall.",
  "greeting": "Hello, Sam from the Shire"
}
```

790 which verifies that the paramters bindings of the values (i.e., "Sam" (name), "the Shire" (place), '13'
791 (age) and '1.2' (height)) on the Trigger were passed to the Action's corresponding input parameters
792 correctly.

793 Discussion

- 794 • Firing the 'meetPerson' Trigger correctly causes a series of non-blocking "activations" of the associated
795 'meetPersonRule' Rule and subsequently the 'hello_world_triggerrule' Action.
- 796 • The Trigger's parameter bindings were correctly passed to the corresponding input parameters on the
797 'hello_world_triggerrule' Action when "firing" the Trigger.

798 Example 6: Using a Deployment file to bind Trigger parameters

799 This example builds on the previous Trigger-Rule example and will demonstrate how to use a
800 Deployment File to bind values for a Trigger's input parameters when applied against a compatible
801 Manifest File

802 Manifest File

803 Let's use a variant of the Manifest file from the previous Trigger Rule example; however, we will
804 leave the parameters on the 'meetPerson' Trigger unbound and only with type declarations.

```
package:
  name: hello_world_package
  ... # Package keys omitted for brevity
actions:
  hello_world_triggerrule:
    function: src/hello_plus.js
    runtime: nodejs
    inputs:
      name: string
      place: string
      children: integer
      height: float
```

```

    outputs:
      greeting: string
      details: string

  triggers:
    meetPerson:
      inputs:
        name: string
        place: string
        children: integer
        height: float

  rules:
    meetPersonRule:
      trigger: meetPerson
      action: hello_world_triggerrule

```

805 **Deployment File**

806 Let's create a compatible Deployment file that will contain the parameter bindings for the input
 807 parameters of the compatible 'hello_world_triggerrule' Action.

```

package:
  hello_world_package:
    triggers:
      meetPerson:
        inputs:
          name: Sam
          place: the Shire.
          children: 13
          height: 1.2

```

808

809 **Github feed**

810 This example will install a feed to fire a trigger when there is activity in a specified GitHub repository.

811 **Manifest File**

```

git_webhook:
  version: 1.0
  license: Apache-2.0
  feeds:
    webhook_feed:
      version: 1.0
      function: github/webhook.js
      runtime: nodejs@6
      inputs:
        username:
          type: string
          description: github username
        repository:
          type: string
          description: url of github repository

```

```

    accessToken:
      type: string
      description: GitHub personal access token
    events:
      type: string
      description: the github event type

    triggers:
      webhook_trigger:
        action: webhook_feed

```

812 **Deployment File**

```

packages:
  git_webhook:
    triggers:
      webhook_trigger:
        inputs:
          username: daisy
          repository: https://github.com/openwhisk/wsktool.git
          accessToken:
          events:push

```

813

814 **Advanced examples**

815 **Github feed advanced**

816 This use case uses the Github feed to create a trigger. When there is any push event, it will send a
 817 notification email.

818 **Manifest File**

```

git_webhook:
  version: 1.0
  license: Apache-2.0
  action:
    emailNotifier:
      version: 1.0
      function: src/sendemail.js
      runtime: nodejs
      inputs:
        email: string
        title: string
  rules:
    githubNotifier:
      trigger: webhook_trigger
      action: emailNotifier

```

819 **Deployment File**

```

packages:

```



```

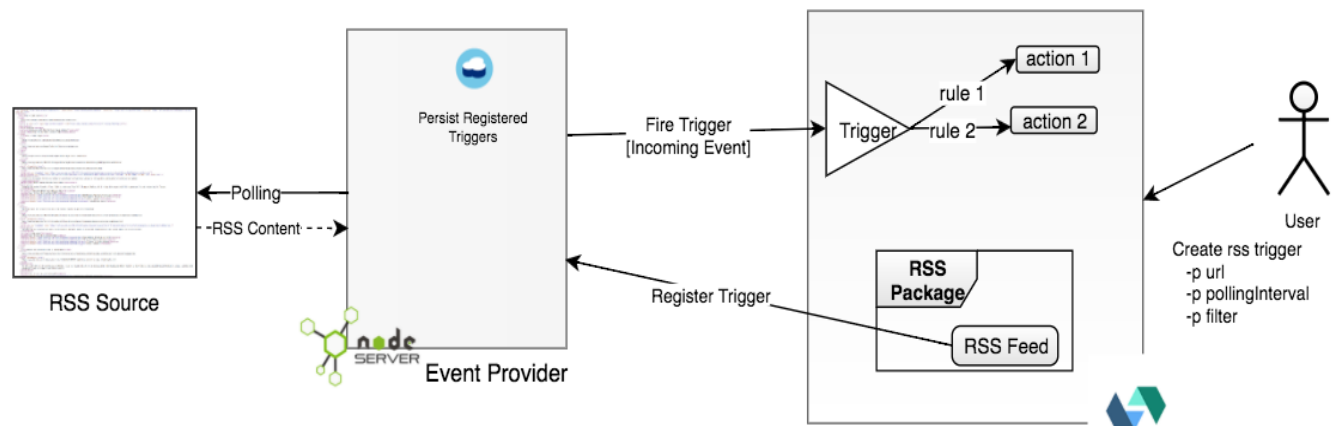
git_webhook:
  feeds:
    webhook_feed:
      inputs:
        email: daisy@company.com
        title: Github Push Notification

```

RSS Package

The RSS package provides RSS/ATOM feeds which can receive events when a new feed item is available. It also defines a trigger to listen to a specific RSS feed. It describes the OpenWhisk package

reposited here:
<https://github.com/openwhisk/openwhisk-package-rss>.



Manifest File

with inline values (no Deployment File)

This example makes use of in-line "values" where the developer does not intend to use a separate Deployment file:

```

rss:
  version: 1.0
  license: Apache-2
  description: RSS Feed package
  inputs:
    provider_endpoint:
      value: http://localhost:8080/rss
      type: string
      description: Feed provider endpoint

  feeds:
    rss_feed:
      version: 1.0
      function: feeds/feed.js
      runtime: nodejs@6
      inputs:

```

```

    url:
      type: string
      description: url to RSS feed
      value: http://rss.nytimes.com/services/xml/rss/nyt/HomePage.xml
    pollingInterval:
      type: string
      description: Interval at which polling is performed
      value: 2h
    filter:
      type: string
      description: Comma separated list of keywords to filter on

    triggers:
      rss_trigger:
        action: rss_feed

```

832

833 **Deployment File**

834 Alternatively, a Deployment File could have provided the same values (bindings) in this way:

```

packages:
  rss:
    inputs:
      provider_endpoint: http://localhost:8080/rss

    feeds:
      rss_feed:
        inputs:
          url: http://rss.nytimes.com/services/xml/rss/nyt/HomePage.xml
          pollingInterval: 2h

```

835

836 Using such a deployment file, allows for more flexibility and the resulting Manifest file would not have
 837 needed any 'value' fields.

838 **Polygon Tracking**

839 This use case describes a microservice composition using Cloudant and a Push Notification service to
 840 enable location tracking for a mobile application. The composition uses Cloudant to store polygons that
 841 describe regions of interests, and the latest known location of a mobile user. When either the polygon set
 842 or location set gets updated, we use the Cloudant Geo capabilities to quickly determine if the new item
 843 satisfies a geo query like "is covered by" or "is contained in". If so, a push notification is sent to the user.

844 **Manifest File:**

```

application:
  name: PolygonTracking
  namespace: polytrack

packages:
  polytrack:

    triggers:

```

```
pointUpdate:
  <feed>

polygonUpdate:
  <feed>

actions:
  superpush:
    inputs:
      appId: string
      appSecret: string

  pointGeoQuery:
    inputs:
      username: string
      password: string
      host: string
      dbName: string
      ddoc: string
      iName: string
      relation: string
    outputs:
      cloudantResp: json

  createPushParamsFromPointUpdate:
    <mapper>

  polygonGeoQuery:
    inputs:
      username: string
      password: string
      host: string
      dbName: string
      ddoc: string
      iName: string
      relation: string
    outputs:
      cloudantResp: json

  createPushParamsFromPolygonUpdate:
    <mapper>

Rules:
  whenPointUpdate:
    trigger:
      pointUpdate
    action:
      handlePointUpdate
  whenPolygonUpdate:
    trigger:
      polygonUpdate
    action:
      handlePolygonUpdate
```

```
    Composition:
      handlePolygonUpdate:
        sequence:
createGeoQueryFromPolygonUpdate,polygonGeoQuery,createPushParamsFromPolygo
nUpdate,superpush
```

845 **Deployment File:**

```
application:
  name: PolygonTracking
  namespace: polytrack

packages:

  myCloudant:
    <bind to Cloudant at whisk.system/Cloudant>

  polytrack:
    credential: ABDCF
    inputs:
      PUSHAPPID=12345
      PUSHAPPSECRET=987654
      COVEREDBY='covered_by'
      COVERS='covers'
      DESIGNDOC='geodd'
      GEOIDX='geoidx'
      CLOUDANT_username=myname
      CLOUDANT_password=mypassword
      CLOUDANT_host=myhost.cloudant.com
      POLYDB=weatherpolygons
      USERLOCDB=userlocation

    triggers:
      pointUpdate:
        <feed>
        inputs:
          dbname: $USERLOCALDB
          includeDoc: true
      polygonUpdate:
        <feed>
        inputs:
          dbname: $USERLOCDB
          includeDoc: true

    actions:
      superpush:
        inputs:
          appId: $PUSHAPPID
          appSecret: $PUSHAPPSECRET
```

```

pointGeoQuery:
  inputs:
    designDoc: $DESIGNDOC
    indexName: $GEOIDX
    relation: $COVEREDBY
    username: $CLOUDANT_username
    password: $CLOUDANT_password
    host: $CLOUDANT_host
    dbName: $POLYDB
polygonGeoQuery:
  inputs:
    designDoc: $DESIGNDOC
    indexName: $GEOIDX
    relation: $COVERS
    username: $CLOUDANT_username
    password: $CLOUDANT_password
    host: $CLOUDANT_host
    dbName: $POLYDB

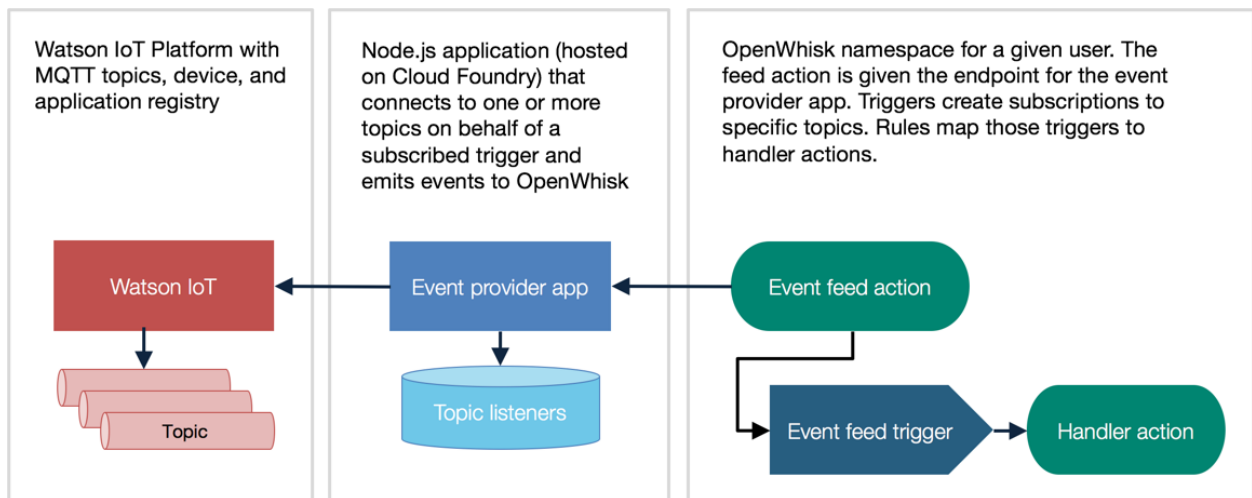
```

846

847 MQTT Package (tailored for Watson IoT)

848 The MQTT package that integrates with Watson IoT provides message topic feeds which can receive
 849 events when a message is published. It also defines a trigger to listen to a specific MQTT topic. It
 850 describes the OpenWhisk package reposited here: [https://github.com/krook/openwhisk-package-mqtt-](https://github.com/krook/openwhisk-package-mqtt-watson)
 851 [watson](https://github.com/krook/openwhisk-package-mqtt-watson).

852



853

854

855 Manifest File

856 *with inline values (no Deployment File)*

857 This example makes use of in-line "values" where the developer does not intend to use a separate
 858 Deployment file:

```

mqtt_watson:
  version: 1.0
  license: Apache-2
  description: MQTT Feed package for Watson IoT
  inputs:
    provider_endpoint:
      value: http://localhost:8080/mqtt-watson
      type: string
      description: Feed provider endpoint

  feeds:
    mqtt_watson_feed:
      version: 1.0
      function: feeds/feed-action.js
      runtime: nodejs@6
      inputs:
        url:
          type: string
          description: URL to Watson IoT MQTT feed
          value: ssl://a-123xyz.messaging.internetofthings.ibmcloud.com:8883
        topic:
          type: string
          description: Topic subscription
          value: iot-2/type/+/id/+/evt/+/fmt/json
        apiKey:
          type: string
          description: Watson IoT API key
          value: a-123xyz
        apiToken:
          type: string
          description: Watson IoT API token
          value: +-derpbog
        client:
          type: string
          description: Application client id
          value: a:12e45g:mqttapp

  triggers:
    mqtt_watson_trigger:
      action: mqtt_watson_feed

```

859

860 *Deployment File*

861 Alternatively, a Deployment File could have provided the same values (bindings) in this way:

```

packages:
  mqtt_watson:
    inputs:
      provider_endpoint: http://localhost:8080/mqtt-watson

  feeds:
    mqtt_watson_feed:

```

```

inputs:
url: ssl://a-123xyz.messaging.internetofthings.ibmcloud.com:8883
topic: iot-2/type/+/id/+/evt/+/fmt/json
apiKey: a-123xyz
apiToken: +-derpbog
client: a:12e45g:mqttapp

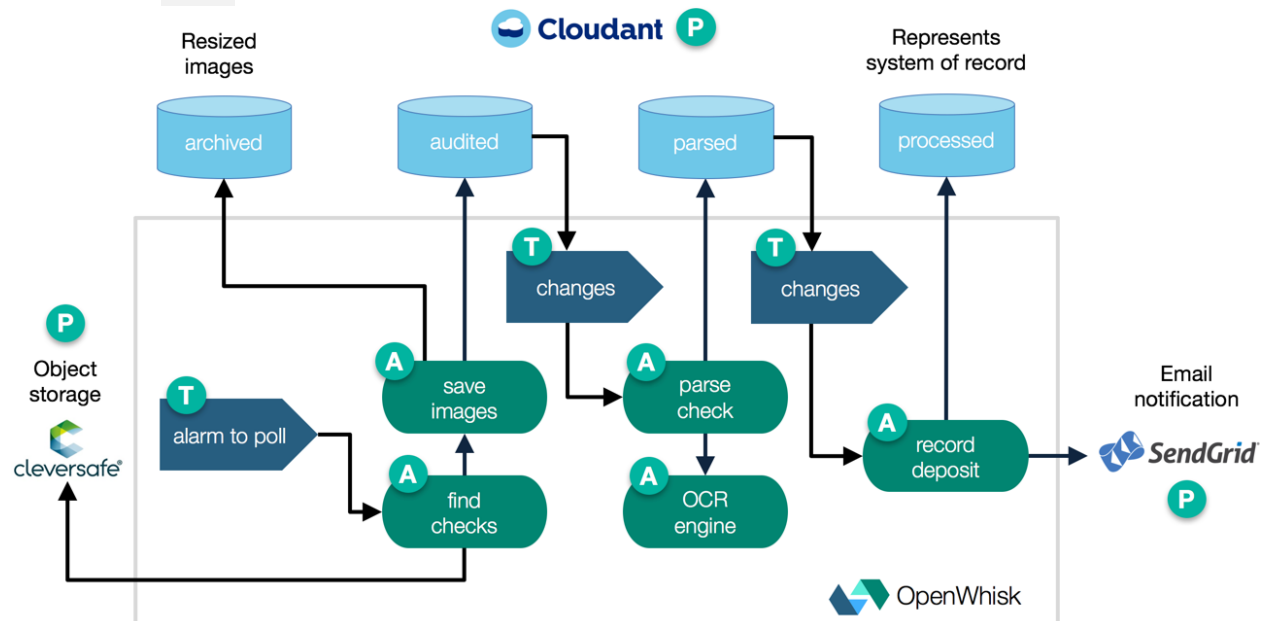
```

Using such a deployment file, allows for more flexibility and the resulting Manifest file would not have needed any 'value' fields.

Check deposit processing with optical character recognition

This use case demonstrates an event-driven architecture that processes the deposit of checks to a bank account using optical character recognition. It relies on Cloudant and SoftLayer Object Storage. On premises, it could use CouchDB and OpenStack Swift. Other storage services could include FileNet or Cleversafe. Tesseract provides the OCR library.

This application uses a set of actions and triggers linked by rules to process images that are added to an object storage service. When new checks are detected a workflow downloads, resizes, archives, and reads the checks then it invokes an external system to handle the transaction.



```
application:
  name: OpenChecks
  namespace: openchecks

packages:
  openchecks:

triggers:
  poll-for-incoming-checks:
    inputs:
      cron: string
      maxTriggers: integer

  check-ready-to-scan:
    inputs:
      dbname: string
      includDocs: boolean

  check-ready-for-deposit:
    inputs:
      dbname: string
      includDocs: boolean

actions:
  find-new-checks:
    inputs:
      CLOUDANT_USER: string
      CLOUDANT_PASS: string
      SWIFT_USER_ID: string
      SWIFT_PASSWORD: string
      SWIFT_PROJECT_ID: string
      SWIFT_REGION_NAME: string
      SWIFT_INCOMING_CONTAINER_NAME: string
      CURRENT_NAMESPACE: string

  save-check-images:
    inputs:
      CLOUDANT_USER: string
      CLOUDANT_PASS: string
      CLOUDANT_ARCHIVED_DATABASE: string
      CLOUDANT_AUDITED_DATABASE: string
      SWIFT_USER_ID: string
      SWIFT_PASSWORD: string
      SWIFT_PROJECT_ID: string
      SWIFT_REGION_NAME: string
      SWIFT_INCOMING_CONTAINER_NAME: string

  parse-check-data:
    inputs:
      CLOUDANT_USER: string
      CLOUDANT_PASS: string
      CLOUDANT_AUDITED_DATABASE: string
      CLOUDANT_PARSED_DATABASE: string
```



```

    CURRENT_NAMESPACE: string

record-check-deposit:
  inputs:
    CLOUDANT_USER: string
    CLOUDANT_PASS: string
    CLOUDANT_PARSED_DATABASE: string
    CLOUDANT_PROCESSED_DATABASE: string
    CURRENT_NAMESPACE: string
    SENDGRID_API_KEY: string
    SENDGRID_FROM_ADDRESS: string

parse-check-with-ocr:
  inputs:
    CLOUDANT_USER: string
    CLOUDANT_PASS: string
    CLOUDANT_AUDITED_DATABASE: string
    id: string
  outputs:
    result: JSON

rules:
  fetch-checks:
    trigger:
      poll-for-incoming-checks
    action:
      find-new-checks
  scan-checks:
    trigger:
      check-ready-to-scan
    action:
      parse-check-data
  deposit-checks:
    trigger:
      check-ready-for-deposit
    action:
      record-check-deposit

```

878 *Deployment File:*

```

application:
  name: OpenChecks
  namespace: openchecks

packages:

  myCloudant:
    <bind to Cloudant at whisk.system/Cloudant>

  openchecks:
    credential: ABDCF
    inputs:
      XXX=YYY

```

```
triggers:
  poll-for-incoming-checks:
    <feed>
    inputs:
      cron: */20 * * * * *
      maxTriggers: 90
  check-ready-to-scan:
    <feed>
    inputs:
      dbname: audit
      includeDoc: true
  check-ready-for-deposit:
    <feed>
    inputs:
      dbname: parsed
      includeDoc: true

actions:
  find-new-checks:
    inputs:
      CLOUDANT_USER: 123abc
      CLOUDANT_PASS: 123abc
      SWIFT_USER_ID: 123abc
      SWIFT_PASSWORD: 123abc
      SWIFT_PROJECT_ID: 123abc
      SWIFT_REGION_NAME: northeast
      SWIFT_INCOMING_CONTAINER_NAME: incoming
      CURRENT_NAMESPACE: user_dev
  save-check-images:
    inputs:
      CLOUDANT_USER: 123abc
      CLOUDANT_PASS: 123abc
      CLOUDANT_ARCHIVED_DATABASE: archived
      CLOUDANT_AUDITED_DATABASE: audited
      SWIFT_USER_ID: 123abc
      SWIFT_PASSWORD: 123abc
      SWIFT_PROJECT_ID: 123abc
      SWIFT_REGION_NAME: northeast
      SWIFT_INCOMING_CONTAINER_NAME: container_name
  parse-check-data:
    inputs:
      CLOUDANT_USER: 123abc
      CLOUDANT_PASS: 123abc
      CLOUDANT_AUDITED_DATABASE: audited
      CLOUDANT_PARSED_DATABASE: parsed
      CURRENT_NAMESPACE: user_dev
  record-check-deposit:
    inputs:
```

```
    CLOUDANT_USER: 123abc
    CLOUDANT_PASS: 123abc
    CLOUDANT_PARSED_DATABASE: parsed
    CLOUDANT_PROCESSED_DATABASE: processed
    CURRENT_NAMESPACE: user_dev
    SENDGRID_API_KEY: 123abc
    SENDGRID_FROM_ADDRESS: user@example.org
  parse-check-with-ocr:
    inputs:
      CLOUDANT_USER: 123abc
      CLOUDANT_PASS: 123abc
      CLOUDANT_AUDITED_DATABASE: audited
      id: 123abc
```

Event Sources

OpenWhisk is designed to work with any Event Source, either directly via published APIs from the Event Source's service or indirectly through Feed services that act as an Event Source on behalf of a service. This section documents some of these Event Sources and/or Feeds using this specification's schema.

Curated Feeds

The following Feeds are supported by the Apache OpenWhisk platform. They are considered "curated" since they are maintained alongside the Apache OpenWhisk open source code to guarantee compatibility. More information on curated feeds can be found here: <https://github.com/apache/incubator-openwhisk/blob/master/docs/feeds.md>.

Alarms

The `/whisk.system/alarms` package can be used to fire a trigger at a specified frequency. This is useful for setting up recurring jobs or tasks, such as invoking a system backup action every hour.

Package Manifest

The "alarms" Package Manifest would appear as follows:

```
# shared system package providing the alarms feed action
alarms:
  version: 1.0
  license: Apache-2
  description: Alarms and periodic utility

  actions:
    alarm:
      function: action/alarm.js
      description: Fire trigger when alarm occurs
      feed: true
      inputs:
        package_endpoint:
          type: string
          description: The alarm provider endpoint with port
        cron:
          type: string
          description: UNIX crontab syntax for firing trigger in
            Coordinated Universal Time (UTC).
          required: true
        trigger_payload:
          type: object
          description: The payload to pass to the Trigger, varies
          required: false
        maxTriggers:
          type: integer
          default: 1000
          required: false

  feeds:
```

```
location: TBD
credential: TBD
operations:
  CREATE:
    TBD
  DELETE:
    TBD
action: alarm
```

895
896

897 **Cloudant**

898 The `/whisk.system/cloudant` package enables you to work with a Cloudant database. It includes the
899 following actions and feeds.

900 *Package Manifest*

901 The “cloudant” Package Manifest would appear as follows:

TBD

902 **Public Sources**

903 The following examples are Event Sources that can provide event data to OpenWhisk. We describe them
904 here using this specification’s schema.

905 **GitHub WebHook**

906 Note: the GitHub WebHook is documented here: <https://developer.github.com/webhooks/>.
907

908 A sample description of the GitHub Event Source and its “create hook” API would appear as follows:

TBD

909

910 Other Considerations

911 Tooling interaction

912 Using package manifest directly from GitHub

913 GitHub is an acknowledged as a popular repository for open source projects which may include
914 OpenWhisk Packages along with code for Actions and Feeds. It is easily envisioned that the Package
915 Manifest will commonly reference GitHub as a source for these artifacts; this specification will consider
916 Github as being covered by the general Catalog use case.

917 Using package manifest in archive (e.g., ZIP) file

918 Compressed packaging, including popular ZIP tools, is a common occurrence for popular distribution of
919 code which we envision will work well with OpenWhisk Packages; however, at this time, there is no
920 formal description of its use or interaction. We leave this for future consideration.

921 Simplification of WebHook Integration

922 Using RESTify

923 One possible instance of a lightweight framework to build REST APIs in Nodejs to export WebHook
924 functionality. See <https://www.npmjs.com/package/restify>
925 RESTify (over Express) provides help in the areas of versioning, error handling (retry, abort) and content-
926 negotiation. It also provides built in DTrace probes that identify application performance problems.

927 Enablement of Debugging for DevOps

928 Isolating and debugging “bad” Actions using (local) Docker

929 Simulate Inputs at time of an Action failure/error condition, isolate it and run it in a “debug” mode.

930

931 Considerations include, but are not limited to:

- 932 • Isolation on separate “debug” container
- 933 • Recreates “inputs” at time of failure
- 934 • Possibly recreates message queue state
- 935 • Provides additional stacktrace output
- 936 • Provides means to enable “debug” trace output
- 937 • Connectivity to “other” debug tooling

938 Using software debugging (LLDB) frameworks

939 This is a topic for future use cases and integrations. Specifically, working with LLDB frameworks will be
940 considered. See <http://lldb.llvm.org/>.

941

942 **Acknowledgements**

943 Thanks to the following individuals who have contributed to the contents:

944

945 Castro, Paul

946 Desai, Priti

947 Guo, Ying Chun

948 Hou, Vincent

949 Krook, Daniel

950 Linnemeier, Micah

951 Liu, David

952 Mitchell, Nick

953 Ortelt, Thomas

954 Rutkowski, Matt

955 Santana, Carlos

956 Villard, Lionel