Scale your Developers with Swagger

What is Swagger?

Swagger is a specification and complete framework implementation for describing, producing, consuming, and visualizing RESTful web services. The overarching goal of Swagger is to enable client and documentation systems to update at the same pace as the server. The documentation of methods, parameters and models are tightly integrated into the server code, allowing APIs to always stay in sync.

With Swagger, deploying managing, and using powerful APIs has never been easier.

Who is responsible for Swagger?

Both the specification and framework implementation are initiatives from Wordnik. Swagger was developed for Wordnik's own use during the development of *http://developer.wordnik.com* and the underlying *http://api.wordnik.com*. Swagger development began in early 2010—the framework being released is currently used by Wordnik's APIs, which power both internal and external API clients.

Why is Swagger useful?

The Swagger framework simultaneously addresses server, client, and documentation/sandbox needs for REST APIs. As a specification, it is language-agnostic. It also provides a long runway into new technologies and protocols beyond HTTP.

With Swagger's declarative resource specification, clients can understand and consume services without knowledge of server implementation or access to the server code. The Swagger UI framework allows both developers and non-developers to interact with the API in a sandbox UI that gives clear insight into how the API responds to parameters and options.

Swagger happily speaks both JSON and XML, with additional formats in the works.

How is Swagger implemented?

As a specification, Swagger is language-agnostic. But since a spec without a usable implementation has limited immediate value, Wordnik has released Swagger implementations in HTML5, Scala, and Java. Client generators are currently available for Scala, Java, Javascript, Ruby, PHP, and Actionscript 3. More client support is underway.

What is Swagger not trying to solve?

Swagger does not currently include a suggestion for supporting multiple API versions from a client or server point of view—versioning information (both of the spec and the underlying API implementation) is declared.

It does not tell you how to write your APIs. For example you can choose to delete an object from your system either by an HTTP DELETE operation or via HTTP GET with query param. While being a good, RESTful citizen is encouraged (and rewarded by an effective API sandbox client), it is not a prerequisite to use Swagger.

Swagger is not trying to solve all problems for all APIs—there will be use-cases that fall outside of the Swagger specification. For this reason, Swagger is an improvement on existing specs like WSDL 2.0 and WADL which must support legacy systems in order to be generally accepted.

Swagger Specification

Core to Swagger is the specification of RESTful web services. This includes three major components: *resource discovery*, *resource declaration* and input/output *model declaration*. Let's explore each component in more detail.

Resource Discovery

The resource discovery mechanism is akin to a sitemap for your web services—based on access (covered below), API consumers are presented with a list of available resources. Both the client generator and the UI documentation sandbox start with the resource discovery URL, which shows "where to go" for more information.

Resource Declaration

At the heart of Swagger is the resource declaration. This describes the APIs provided by the system and the required and optional fields for each operation. The resource declaration also provides information on the allowable inputs and describes complex objects transmitted to and from the API, as well as the error conditions that can occur if something goes wrong.

A sample Resource Declaration is shown in Appendix B and includes the following:

basePath. The fixed portion of the API being declared. All API paths are relative to this basePath.

swaggerVersion. The version of the Swagger specification implemented by this provider.

apiVersion. The version of the implemented API.

apis. An array of APIs available to the client.

models. An array of models used in this API.

APIs

Each element in the APIs array is a logical grouping of possible operations on a fixed path. For example a *GET* and *DELETE* operation against the same URL would be described in a single API element.

Parameters for each operation come in a number of flavors, which correspond to typical REST patterns. They are named in the *name* attribute and are described by the *paramType* attribute. The allowable *paramType* values are:

path. The input is part of the URL itself, specifically the URL portion which corresponds to the {*name*} in the API path. All path parameters are mandatory and must be non-zero length

query. The input is specified by a key/value query param in the form of {*key*}={*value*}. Multiple query parameters are separated by "*&*" delimiters

post. One and only one input object is supplied as part of a HTTP post operation. This is the only supported mechanism for supplying complex objects to the operations

Other self-explanatory attributes of the API object:

dataType. The primitive type (see
Appendix A) of the allowable input values for query and path params. For post params, complex datatypes are supported per the *models* section. Even though query and path params are passed as part of the URI, specifying the type (Integer, String) can help the client send appropriate values.
responseClass. This declares the class of the model being returned from the API operation. See Appendix B.

allowableValues. For query parameters, this outlines the options that the client can supply. The Swagger framework does not enforce that the client conforms to the allowable values—that is the job of the underlying server implementation.

allowMultiple. For query parameters, this signals to the client that multiple values can be specified in a comma-separated list.

Parameter Attributes			
paramType	path	query	post
allowableValues	optional	optional	n/a
allowMultiple	n/a	optional	n/a
dataType	required	required	required
description	optional	optional	optional
name	required	required	optional
required	always true	optional	optional

Figure 1. Table of possible attributes for the API object

errorResponses

Each API has a corresponding array of errorResponse objects. These describe the possible error codes and what they mean to the client (note: a HTTP 500 error response is always possible and implicit).

Error response objects contain the following attributes:

code. The HTTP error code returned. See *http://en.wikipedia.org/wiki/List_of_HTTP_status_codes* for a complete list.

reason. A human-readable reason for the error, applicable to this operation. Note that returning a 404 error response will tell you "not found" but for usability, the error code should be more explicit with reasons like "user was not found" to avoid confusion between general service availability and application logic.

It is important to note that HTTP as a protocol is somewhat abstracted from the framework. That said, Swagger offers an exception mapper which will catch application-level exceptions and map them to HTTP responses.

models

The models array describes complex objects which can be passed to and from the API. In the description of each object there is no effort put into describing an entire object graph—instead complex sub-objects have named types, which each have their own model description. All models will eventually devolve into primitives.

The model description follows draft 3 of the JSON schema as submitted to the IETF. For more information please see: *http://tools.ietf.org/html/draft-zyp-json-schema-03*

Wordnik Swagger Framework Implementation

The Swagger specification is fully implemented by Wordnik and available to open source as a series of tools and libraries. See *http://swagger.wordnik.com* for links to the full source code.

Server

Core to Swagger is the goal of having a server-driven API framework. It is implemented by an OSS server library and depends on a number of OSS projects. The core requirements for integrating include:

Java runtime. Version 1.6 or greater is required—internal testing uses the Oracle JVM runtime from *http://java.oracle.com*.

Scala runtime. Version 2.8.1 final or greater is required. Scala is available from *http://www.scala-lang.org/*

Jersey framework. Version 1.7 or greater of the JAX-RS framework is required.

Jackson JSON parser. Version 1.7.7 or greater of the Jackson JSON library is required.

The Swagger server is built using *ant* and *ivy*. Both Apache projects are available from *http://apache.org*.

The Swagger specification is achieved by annotating JAX-RS-aware classes with a small set of custom annotations. These declare the endpoints, parameters, and return types. In addition, the decoration of models with *JAXB* annotations makes them automatically discoverable and describable by the Swagger framework.

Since Swagger supports both XML and JSON formats, all examples declare their format in the resource itself. While not required by the specification—desired format can be supplied in request headers—it is strongly encouraged that the system have explicit paths for XML and JSON. You will continue to see this style throughout this documentation and on all Swagger-related examples either explicitly or with a *.{format}* placeholder.

Resource Discovery

Resources are declared automatically by the *com.wordnik.swagr.sandbox.resource.ApiListingResource*. This will tell the Swagger framework to introspect and instrument all JAX-RS instrumented classes specified in the *com.sun.jersey.config.property.packages* parameter in the *web.xml*. See Appendix D for an example.

Once instrumented, a resource listing is automatically made available in the {*ws-context*}/*resources*.{*format*}

Resource Annotations

The following are required for proper code annotation:

@Api. This tells the framework about a top-level resource (i.e. api.wordnik.com/v4/{resource}.{format}). This differs from the @Path annotation—this is to allow for the same instrumentation to be supplied for a number of request and response formats. So the resources */v4/words.json* and */v4/words.xml* would be annotated as *@Api("/words")*.

The *@Api* annotation supports the following attributes:

value. The short description of what the API does.

description. A longer description of what the API does.

Operation Annotations

@ApiOperation. This annotation allows for the instrumentation of a single API operation. The annotation also considers the following method-level annotations:

@GET/*@POST*/*@PUT*/*@DELETE*. These *JAX-RS* annotations indicate what type of HTTP operation the method is.

@Path. Also a *JAX-RS* annotation, this indicates the URI relative to the base path in the *@Api* annotation.

@ApiErrors. This is an array of *@ApiError* annotations (below) which describe possible error conditions for the operation.

@ApiError. This describes a specific error *reason* and HTTP response *code*.

In addition, the *@ApiOperation* supports the following attributes:

value. This is the human-readable description of what the API does.

responseClass. The fully-qualified response model to be returned from the operation. For instrumentation, this requires the full package specifier. For primitives, simply specify any one of the values listed in Appendix A.
multiValueResponse. This indicates that the response contains an array of *responseClass* objects.

tags. This is a comma-separated list of tag strings which can be used to group operations. Tags are optional in Swagger.

Parameter Annotations

Following the JAX-RS style annotations, the input values to methods are annotated with Swagger annotations to construct the schema. Specifically:

@ApiParam. This is an annotation on a particular method input value. It supports the following values:

value. This is the human-readable description of what the method does.

required. For post & query params (see full compatibility table in Figure 1), this can indicate whether the value is optional or mandatory.

allowMultiple. For query params, indicates whether multiple values are allowed (choice vs. option).

allowableValues. Indicates supported, allowable values for the parameter

See Appendix D for an example.

Security

Swagger provides a general mechanism for API security. Based on a filter, the server can be implemented with virtually any type of security. Note the security implementation is left for the API developer to create. The input to the filter interface is:

apiPath:String. The full path to the method being called.

headers:HttpHeaders. All headers supplied in the request to the api as detected by the *JAX-RS* implementation.

uriInfo:UriInfo. Additional *URI* information provided by the *JAX-RS* implementation.

This mechanism secures *resources* based on *operations* being requested. Object-level security should be implemented "on the other side" of Swagger—namely in the API layer of the application.

There is a sample implementation of this interface in the sample app which demonstrates key-based access to resources. See *com.wordnik.swagger.sample.util.ApiAuthorizationFilterImpl*.

Client Library Generator

Since all Swagger-compliant APIs expose a Resource Declaration, client libraries can be automatically generated for any language. The client generator can read both resource methods and models to generate a fully-capable REST client.

The Client Library Generator utilizes the Antlr String Template Library (*http://www.stringtemplate.org/*) to process a small number of code templates against the response of the Resource Declaration query against the server. The output from the generator is a light-weight client library for use in a number of target languages.

In addition, the Client Library Generator includes a test framework for running client-side tests against a Swagger-compatible server. This includes response checking as well as a template for verifying expected responses.

Developer Site/API Sandbox

The Swagger framework includes an HTML5 + javascript-based API Sandbox. The tool needs only a Resource Discovery URL (i.e. *http://api.wordnik.com/v4/resources.json*) and an API key. The tool can then be used to browse and test any API which implements the Swagger specification.

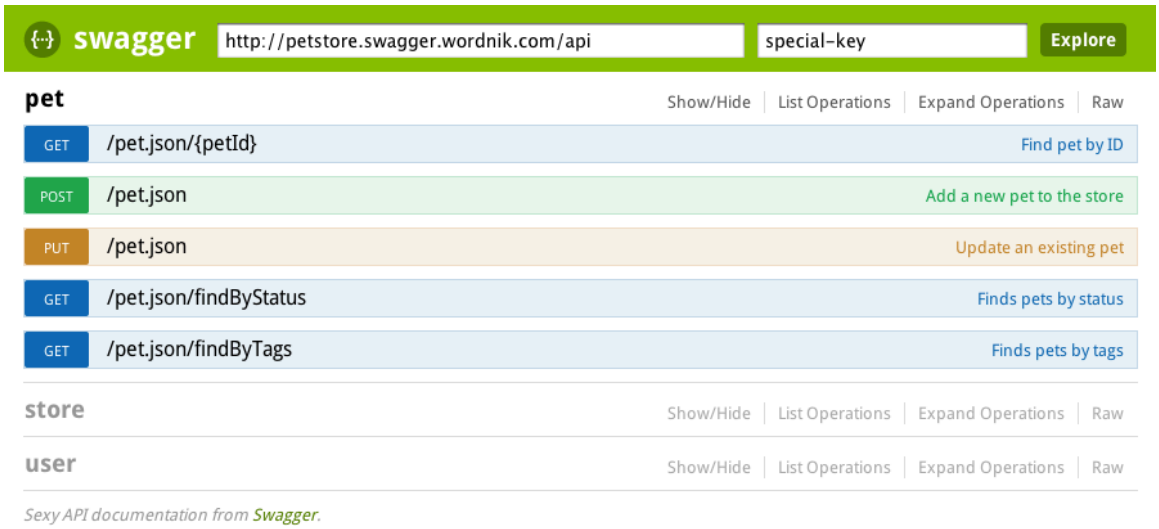

Figure 2. Sample Swagger API.

The sandbox is useful for understanding the routes that the particular API key has access to. Filtering of resources is applied automatically to any resource list based on an implementation of the *ApiAuthorizationFilter*. So your users will only see their resources. Each method appears in its own line in the UI with color coding & grouping based on the HTTP method type.

The UI stores the Resource Discovery URL and API key in the browser's local storage. The UI can be delivered to developers as either a local application or by deploying it on a remote server.

As multiple resources are encountered the UI will group them appropriately, see the */pet* and */store* resource examples in Figure 2.

Currently only GET is supported with the sandbox.

Appendix A

Allowable primitive datatypes

String

Integer

Long

Double

Boolean (true, false, yes, no are all treated case-insensitive)

Note: Dates should be described as String values in ISO-8601 format and de-serialized using the convenience methods in the server implementation.

Appendix B

Sample Model Response from the sample resource declaration at *http://localhost:8002/api/pet.json*

The embedded "category" object is declared elsewhere in the model tree. Model declarations do not include sub-objects.


```
"models":{
    "category":{
        "properties":{
            "id":{
                "type":"long"
            },
            "name":{
                "type":"string"
            }
        },
        "id":"category"
    },
    "pet":{
        "properties":{
            "tags":{
                "type":"array",
                "items":{
                    "$ref":"tag"
                }
            },
            "id":{
                "type":"long"
            },
            "category":{
                "type":"category"
            },
            "status":{
                "type":"string",
                "description":"pet status in the store",
                "enum":[ "available", "pending", "sold" ]
            },
            "name":{
                "type":"string"
            },
            "photoUrls":{
                "type":"array",
                "items":{
                    "type":"string"
                }
            }
        },
        "id":"pet"
    }
}
```

Appendix C

Sample resource listing:

http://localhost:8002/api/resources.json


```
{
    "apis":[
        {
            "path":"/pet.{format}",
            "description":""
        },
        {
            "path":"/user.{format}",
            "description":""
        },
        {
            "path":"/resources.{format}",
            "description":""
        }
    ],
    "basePath":"http://localhost:8002/api",
    "swaggerVersion":"1.0",
    "apiVersion":"0.1"
}
```


Sample operation listing:


```
{
    "apis":[
        {
            "path":"/pet.{format}/{petId}",
            "description":"Operations about Pets",
            "operations":[
                {
                    "parameters":[
                        {
                            "name":"petId",
                            "description":"ID of pet that need to be fetched",
                            "dataType":"string",
                            "required":true,
                            "allowMultiple":false,
                            "paramType":"path"
                        }
                    ],
                    "httpMethod":"GET",
                    "notes":"Returns a pet when id < 10. Id > 1000 or non
integers will simulate API error conditions",
                    "nickname":"getPetById",
                    "responseClass":"pet",
                    "deprecated":false,
                    "summary":"Find Pet by id"
                }
            ],
            "errorResponses":[
```

```
        {
            "reason":"Invalid ID supplied",
            "code":400
        },
        {
            "reason":"Pet not found",
            "code":404
        }
    ]
}
```

Appendix D

Sample resource annotation

```
@Path("/pet.json")
@Api(value = "/pet", description = "Operations about Pets")
@Singleton
@Produces(Array("application/json"))
class PetResourceJSON extends Help
  with PetResource
```

Sample method annotation

```
@GET
@Path("/findByStatus")
@ApiOperation(value = "Finds Pets by status",
  notes = "Multiple status values can be provided with CSV",
  responseClass = "com.wordnik.swagger.sample.model.Pet",
  mutiValueResponse = true)
@ApiErrors(Array(
  new ApiError(code = 400, reason = "Invalid status value")))
def findPetsByStatus(
  @ApiParam(value = "Status values that needs to be considered for filter",
            required = true,
            defaultValue = "available",
            allowableValues = "available,pending,sold",
            allowMultiple = true)
  @QueryParam("status") status: String) =
{
  Response.ok.entity(petData.findPetByStatus(status)).build
}
```