Tcl WebServices: FlightXML at FlightAware

Jonathan Cone
Software Developer
FlightAware

FlightAware is a flight tracking company founded in 2005 with a primarily Tcl code base. The company consumes numerous data feeds from entities such as the FAA, NOAA, Eurocontrol, Australia, New Zealand, its own ADS-B network and many others. The data from these feeds is processed, normalized and filtered before being stored in a PostgreSQL database server and sometimes in an in-memory data store referred to as Birdseye. When the company first started, there was only a website for accessing its aviation data and it quickly became clear that commercial and enthusiast customers needed programmatic ways of accessing FlightAware's data. Over the years FlightAware has launched three versions of an API called FlightXML to provide that access. The scope of the data offered has expanded over the years and now includes items such as weather METAR and TAF data, Airline Flight Schedules, Enroute, Scheduled, Arrived and Departed flight information for airports, extended information on specific flights, tracking data on flights, registered owner for aircraft, and airport delays to name a few.

The first version of FlightXML, originally called DirectFlight, was launched in 2006, shortly following the launch of the main website. At that time, the decision was made to host FlightXML separately from the website so a distinct tclhttp instance was used. Since DirectFlight was hosted separately, it required a different hostname from the website and directflight.flightaware.com was used. The preference would have been to have the service on the main hostname if it had been technically convenient, but that was not the case and the unique hostname has been used to this day. As REST services were not as popular at that time and no tcl libraries supported REST servers, DirectFlight only offered a SOAP service. The TclSOAP library was used to implement the DirectFlight service, using the SOAP::wsdl and SOAP::domain packages. DirectFlight was able to use much of the existing functionality in FlightAware by calling existing procs to generate the requested data and then formatting the response in the service definition of each method. The same is generally still true in the latest versions of FlightXML. In 2010 FlightAware migrated from the tclhttp web server to the more broadly supported Apache webserver allowing the website and flightxml to be served by the same Apache instance. Apache Rivet is now used as the entry point for all FlightXML services. The billing structure for DirectFlight charges customers based on usage and API call class (some calls are more expensive than others), with invoices sent on a monthly basis after usage is computed.

Following the success of DirectFlight, later renamed FlightXML1, a second version of the API was planned. One disadvantage of the TclSOAP library was that it was not very popular and had not been in active development since about 2004. By the time FlightXML2 was under development, tclws was becoming the clear choice for implementing a SOAP service with Tcl. Therefore in 2010, FlightXML2 launched and was built using the tclws library. FlightXML2 expanded the dataset provided through the API to include additional details on flights such as the faFlightID, which uniquely identifies a flight on the FlightAware system. Additional

information on METARs were included and the ability to programmatically setup alerts. In 2011 enhancements were made to tclws to include a REST service as well as the SOAP service.

The latest incarnation of FlightXML, FlightXML3, is currently in beta and the production release is expected to be available at the end of 2017 or beginning of 2018. V3 uses the tclws library to provide the service definitions, but some incremental enhancements have been made to the library. The service definition was updated to support an optional operator. Much as the name implies, that optional operator allows the developer to specify that the response or request field is optional and may not be included. The business motivation for FlightXML3 is to allow users to access more of the data they desire using fewer numbers of API calls. With FlightXML1 and FlightXML2, users would often need to make multiple calls to get all the relevant information on a flight, and some data such as block in and block out (gate arrival and departure) times were not available. For example, in order to reconstruct the data found on the main FlightAware page for an airport using FlightXML2, calls must be made to Departed, Arrived, Enroute and Scheduled and those calls were all combined into a single AirportBoards call in FlightXML3. The billing structure for FlightXML3 has moved to a subscription plan where users can sign up for a block of calls per month with a maximum query rate. Internally the query rate is enforced using memcached.

FlightXML Architecture and Implementation

The architecture for FlightXML2 and 3 is similar and will be the focus of this section. Requests to both www.flightaware.com and flightxml.flightaware.com are first passed to a varnish server responsible for caching web requests. The flightxml domain is excluded from that caching mechanism so requests are always passed back to the Apache webservers. The Apache configuration allows for POST or GET requests, and SOAP or REST requests are routed to the relevant rivet page based on the path. For example, a request to flightxml.flightaware.com/json/FlightXML2/FlightInfoEx will be routed to the rest2.rvt page. The rivet pages load the flightxml, tclws packages and some configuration information, authenticate the user if required, handle the actual request and any errors and finally send the response back. Figure 1 shows the flow of a request from users to FlightXML.

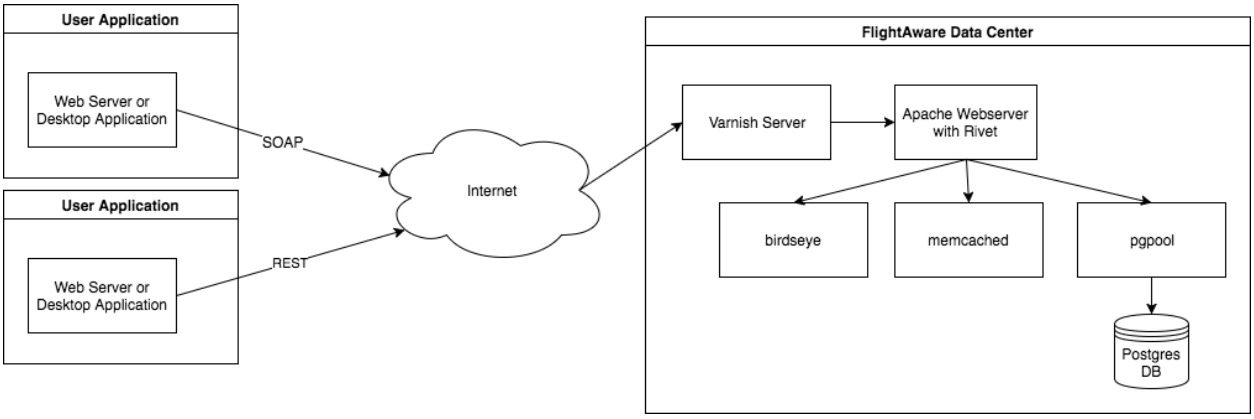

Figure 1 - FlightXML Request

FlightXML encapsulates the service and API method definitions in a Tcl package which the landing pages require. In order to make use of tclws, the FlightXML2 package begins by importing WS::Server and a utility package which contains functionality common to all versions of flightxml. The tclws service definition then defines the details of the service being generated. There are a number of options available when defining the tclws service including host, description, service, author, host, prefix and others. Calling the tclws service definition proc sets up the global serviceArr array with the specified options, installs the wsdl document, installs the wsdl and installs the operations of needed based on the mode of operation. tclws supports running on tclhttpd, Apache with rivet, AOLserver, WUB, wibble and embedded mode. The FlightXML2 service definitions specifies rivet mode.

```
package require flightaware-main
package require WS::Server 2.4
…
set flightxml2_serviceName “FlightXML2”
set ::flightxml2_endpoint_base "/soap/FlightXML2"
set ::flightxml2_hostname "flightxml.flightaware.com"
set ::flightxml2_stylesheet "/commercial/flightxml/flightxml2.css"
…
##
## Define the service
##
::WS::Server::Service \
    -mode         "rivet" \
    -service      $::flightxml2_serviceName \
    -author       {FlightAware} \
    -description  {FlightXML2 Web Services} \
    -htmlhead     {FlightXML2 Web Services} \
    -stylesheet   $::flightxml2_stylesheet \
    -docFormat    "html" \
    -traceEnabled N \
    -host         $::flightxml2_hostname \
    -prefix       $::flightxml2_endpoint_base \
    -errorCallback  fxmlv2_error_logging_callback
```

Figure 2 - tclws Service Definition

Individual operations are defined by calling the ::WS::Server::ServiceProc. The command takes 5 arguments: the service name, the name info which is a list with the operation name, the return type and the description, the argument list which is list of form argument name followed by argument type information, the documentation and finally the implementation body. FlightXML defines the documentation before the service definition in a variable for readability and we have found that works well. When defining arguments the argument type info should be a key value list with the type and optionally the comment. Any comment added here will be included in the generated documentation. Figure 3 is the service definition for the FlightXML2 FlightInfoEx method which returns an array with details on a specified flight.

The return and argument types can be one of four types: Simple non-array, Simple array, Non-Simple nonarray and Non-Simple array. The simple types are mostly what you might

expect and a full listing are available in the tclws documentation, but string, Boolean, decimal, float, double, duration, dateTime, time, date and others are supported. If however your type is not a simple one but rather a struct or nested structs then tclws allows for the creation of custom types. The ::WS::Utils::ServiceTypeDef proc defines new types and accepts arguments for the mode (Client or Server), the service name, the name of the type and the definition. The definition of the type consists of the fields followed by the field information, where the field information is a list of the type, typeName, comment and commentString. The comment is optional and if the type is an array then the typeName should be suffixed with (), for example int() would be an array of integers. Figure 4 shows the type definition for a FlightExStruct and FlightInfoExStruct. The FlightInfoEx operation shown earlier returns a FlightInfoExStruct which contains the next_offset field, an int, and an array of flights of type FlightExStruct. In the WSDL generated by tclws, the minOccurs and maxOccurs will be determined based on whether the type is an array or optional. If a type is optional then the minOccurs will be 0 and if the type is an array the maxOccurs will be infinite. Otherwise both values will be 1. When using a custom type as the return type of an operation, the returned value should be a list consisting of the field name followed by the field value, and the top level list should be the operation name with "Result" appended followed by the results. For example FlightInfoEx returns a list with {FlightInfoExResult {next_offset -1 flights {…} } }.

```
#
# Define FlightInfoEx
#
set dochtml {<p><b>FlightInfoEx</b> returns information about flights
for a specific tail number (e.g., <strong>N12345</strong>), or an ident
(typically an ICAO airline with flight number, e.g.,
<strong>SWA2558</strong>), or a FlightAware-assigned unique flight
identifier (e.g. faFlightID returned by another FlightXML
function).</p>
...additional description
<p>See <a href="#op_FlightInfo">FlightInfo</a> for a simpler
interface.</p>}
# --
::WS::Server::ServiceProc $::flightxml2_serviceName {FlightInfoEx {type
FlightInfoExStruct comment "returned results"}} {
    ident    {type string comment "requested tail number, ..."}
    howMany  {type int comment "maximum number of past flights to
obtain..."}
    offset   {type int comment "must be an integer value of the
offset..."}
} $dochtml {
    # body implementation
    ::log::log notice "FlightInfoEx '$ident' '$howMany' '$offset'"
    ::flightxml::charge
    ...
    return [list FlightInfoExResult $result]
}
```

Figure 3 - tclws Operation Definition

An advantage of using tclws is that it will generate service documentation and WSDL automatically. The –docFormat option allows you to specify if the documentation will be in

plain text (“text) or html (“html”). The ::WS::Server:: generateInfo proc generates the documentation in the specified format and will then output that data based on the mode of operation. For the FlightXML case, the headers are set to 200 with a type of “text/html; charset=UTF-8” and the html or text is output using a puts. Other modes use methods specific to that server, such as the tclhttpd mode which calls ::Httpd_ReturnData. The documentation is generated based on the service definition, operations and complex types defined. As mentioned previously the description for each operation will be included, and inputs will be listed along with any information contained in the comment for the type definition. The return type and its description are part of the operation definition itself. The WSDL is generated by calling the ::WS::Server::generateWsdl proc and will output those results based on the mode of operation. For FlightXML we expose different endpoints for accessing documentation and the wsdl by examining the path to see if the wsdl, doc or jsondoc are set and if so then calling the relevant tclws proc. Figure 5 illustrates how that is done for FlightXML2 on its landing rivet page.

```
::WS::Utils::ServiceTypeDef Server $::flightxml2_serviceName
FlightInfoExStruct {
    next_offset                     {type int}
    flights                         {type FlightExStruct()}
}

::WS::Utils::ServiceTypeDef Server $::flightxml2_serviceName
FlightExStruct {
    faFlightID              {type string comment "unique identifier
                             assigned by FlightAware for this flight"}
    ident                   {type string comment "flight ident or
                             tail"}
    aircrafttype            {type string comment "aircraft type ID"}
    filed_ete               {type string}
    filed_time              {type int}
    filed_departuretime     {type int}
    filed_airspeed_kts      {type int}
    filed_airspeed_mach     {type string}
    filed_altitude          {type int}
    route                   {type string}
    actualdeparturetime     {type int}
    estimatedarrivaltime    {type int}
    actualarrivaltime       {type int}
    diverted                {type string comment "boolean indicator"}
    origin                  {type string comment "the origin airport
                             ID"}
    destination             {type string comment "the destination
                             airport"}
    originName              {type string}
    originCity              {type string}
    destinationName         {type string}
    destinationCity         {type string}
}
```

Figure 4 - tclws Type Definition

```
# Check if user is requesting WSDL or documentation
if {[var exists wsdl] || [env PATH_INFO] == "/wsdl"} {
    # send the WSDL without requiring authentication

    ::WS::Server::generateWsdl $svcname RivetClient
} elseif {[var exists doc] || [env PATH_INFO] == "/doc"} {
    # send the functional documentation without requiring
authentication

    ::WS::Server::generateInfo $svcname RivetClient
} elseif {[var exists jsondoc] || [env PATH_INFO] == "/jsondoc"} {

    ::WS::Server::generateJsonInfo $svcname RivetClient
}
...
```

Figure 5 - rivet page loading WSDL or documentation

FlightXML Component Enhancements

As part of the development of FlightXML2, in 2011 FlightAware added REST service functionality to the tclws service. This development was driven by frequent requests from users for that functionality as REST services grew in popularity and became ubiquitous. Tclws was updated to support the "-rest" argument to the callOperation method. By passing the "-rest" argument, tclws will generate a JSON response to the call using the yajltcl library. This means that any existing tclws services can include a REST service with a fairly minimal code change.

Internally a number of changes were made to implement the REST behavior in tclws. New methods were added to handle conversion from dictionary to a JSON tree for simple types, simple array types, non-simple types and non-simple array types. The conversion between types is limited to a subset of the types available in tclws, but covers the most commonly used ones. Figure 6 shows the type conversion between tclws types and yajl types. When the REST flavor is selected, the Content-Type for the response is also set to "application/json". An additional config option was added to beautify JSON which will format the output in a more readable human readable way. On the FlightAware fork of tclws REST responses would always set the status code to 200 even if an error was encountered. This was later updated, and now if the error code is a valid http status code then tclws will respond with that status code. For example if you had a user who had given you an invalid input, in the body of the operation definition you can throw an error and set the errorCode to 400 to have tclws respond with the 400 status code.

During FlightXML3 development, it was discovered that the WSDL generated by tclws always had a minOccurs of 1. For some SOAP clients, the default behavior is to strictly enforce input parameter and response conformity to the WSDL. In FlightXML not all input parameters are required on every operation, and responses may not contain all the listed elements depending on the query. To address this issue, an optional operator was added to tclws type declarations. The question mark optional operator was adopted, borrowing from Swift, to indicate that a type was optional. When the type is marked the '?', the minOccurs in the WSDL will then be set to 0. Figure 8 shows an example of declaring a non-simple type in tclws with optional fields from a FlightXML3 type declaration.

```
      # mapping of how the simple SOAP types should be serialized
      # using YAJL into JSON.
      array set ::WS::Utils::simpleTypesJson {
          boolean "bool"
          float "number"
          double "double"
          integer "integer"
          int "integer"
          long "integer"
          short "integer"
          byte "integer"
          nonPositiveInteger "integer"
          negativeInteger "integer"
          nonNegativeInteger "integer"
          unsignedLong "integer"
          unsignedInt "integer"
          unsignedShort "integer"
          unsignedByte "integer"
          positiveInteger "integer"
          decimal "number"
      }
```

Figure 6 - simple SOAP types to YAJL

```
if {![flightaware_validateFlightId $faFlightID]} {
    error "INVALID: invalid {faFlightID}" {} 400
}
```

Figure 7 - Passing http status code to tclws through error code

```
# Forecast Wind Struct
::WS::Utils::ServiceTypeDef Server $::flightxml3_serviceName
ForecastWindStruct {
      symbol            {type string comment "Raw TAF wind symbol"}
      direction   {type string comment "Wind direction"}
      speed             {type int comment "Wind speed"}
      units             {type string? comment "Optional. Wind units."}
      peak_gusts  {type int? comment "Optional. Peak gusts for forecast."}
}
```

Figure 8 - tclws type definition with optional operator

With the addition of the optional operators, it was then possible to have tclws strictly enforce input and response parameter requirements. The service configuration now has an option to strictly enforce all required parameters, "-enforceRequired". The option is set to "N" by default, but if enabled then tclws will throw an error if either the request or response is lacking a required parameter.

The initial mapping of the float SOAP type in tclws was to the float yajl type. Because of the limitations of float precision, responses with float types in FlightXML2 frequently had extraneous decimal data. For example if an airport's elevation was 80.5 feet and a user requested the AirportInfo operation, the response would have an elevation of 80.49999998 feet. To resolve

this issue, the float SOAP type was mapped to the yajl number type. Yajl treats a number as a string internally so the response was correct, but it was found that yajltcl did not check to see that a number really was a number and would accept any input for that type. An enhancement was then made to yajltcl to include a lexer in yajltcl to check a numeric input and ensure that it was a valid number.

A final enhancement to tclws as part of FlightXML development added an error handler callback option to the service definition. If the proc name of an error handler is specified in the "-errorCallback" option, then that callback will be called in the event of an error executing the requested operation. The arguments passed to the callback are the errorMsg, the httpStatus code, the operation name and the flavor (REST or SOAP). In FlightXML3 the errorCallback is used to set the http status code of the response based on a parsing of the error message. The error callback could also be used to log those errors to a reporting service. For FlightXML, one server is running a udp listener which logs all errors to an output file as well as sending error information to a zabbix server for graphing. As FlightXML is operating over multiple webservers this provides a central repository for all error related information. In particular this allows FlightAware to monitor for particular errors and alert users if their programs are making repeated failed attempts (FlightXML charges users for invalid queries so it benefits users to know if they are making multiple invalid attempts each day). Figure 9 shows the definition the error callback handler for FlightXML3. To set the response status code, the httpStatus is upvar and then set to the appropriate value based on the type of error.

Future enhancements for FlightXML have proposed supporting a filter parameter on all tclws queries with OData operator support. The proposed changes would implement a subset of OData operators such as: eq, ne, gt, ge, lt, le, and, or not. The operators would allow a user to do serverside filtering of their recordset by operating on the response fields. For example for FlightXML a user who is requested information on flights at Chicago O'Hare may want to only see flights with a destination of John F Kennedy in New York. The filter parameter could then be set to "filter=destination.code eq KJFK" and tclws would filter the results for just flights with a destination of KJFK before transmitting the response.

The aforementioned enhancements to tclws were merged into the public repository in June of 2017, so these features are now available to any tclws consumers. As FlightAware continues its work on Tcl web services any future enhancements will be pushed to the community as well. FlightXML proves that high performance, reliable web services based on Tcl libraries are a viable option for any developers wanting to expose their services and data in an API.

```
#
# Define error handling callback.
#
proc fxmlv3_error_logging_callback {errorMsg httpStatus method {flavor ""}} {

      if {[info exists ::user(username)]} {
            set username $::user(username)
      } else {
            set username {}
      }

      if {$httpStatus ne {}} {
            upvar $httpStatus httpStatusCode
      } else {
            set httpStatusCode {}
      }

      set logMsg "flightxml3 "
      if {[string match -nocase {*UNKNOWN_METHOD*} $errorMsg]} {
            append logMsg "invalidMethod {$errorMsg} $username $method"
            set httpStatusCode 404
      } elseif {[string match -nocase {*BAD_FLAVOR*} $errorMsg]} {
            append logMsg "badFlavor {$errorMsg} $username $method"
            set httpStatusCode 400
      } elseif {[string match -nocase {INVALID_ARGUMENT*} $errorMsg]} {
            append logMsg "invalidArgument {$errorMsg} $username $method"
            set httpStatusCode 400
      } elseif {[string match -nocase {NO_DATA*} $errorMsg]} {
            append logMsg "noData {$errorMsg} $username $method"
      } elseif {[string match -nocase {APP_FAULT*} $errorMsg]} {
            set httpStatusCode 500
            append logMsg "appFault {$errorMsg} $username $method"
      } else {
            append logMsg "unknownError {$errorMsg} $username $method"
            set httpStatusCode 500
      }

      ::flightxml::udp_error_logging $logMsg
}
```

Figure 9 - FlightXML3 error callback handler