tclws versioning support

Author:
Jonathan Cone
Senior Software Developer
FlightAware, LLC

Web Services for Tcl (tclws) is a package providing client and server methods for generating and consuming http web services. The package produces document/literal responses with HTTP SOAP or JSON content depending on configuration. A number of methods are available to define a service, custom data types, documentation and WSDL. At FlightAware, tclws is used to define and operate the FlightXML2 and FlightXML3 services. These are query based services where clients make HTTP requests to obtain flight data. FlightXML2 is a mature product and changes to its data structure or service definition are not allowed at this time as that would cause issues for any clients using the SOAP interface. However some customers have requested changes to those services, so FlightXML2b and 2c are offered (but not advertised) and consist of completely separate service definitions. In some cases a method in FlightXML2 is duplicated in FlightXML2b with some change to the return fields or query logic with some amount of code duplication. tclws previously provided no clear way to expand an existing service with future versions or revisions that isolated the new definitions from previous ones unless a new service was defined. The changes introduced to tclws this past year endeavor to solve that problem and provide a solution where service definitions can be versioned and code duplication minimized.

Before proposing versioning support in tclws, some attempts were made to address versioning for FlightXML3 in a way that would avoid the issues encountered with FlightXML2. An initial attempt would have reloaded the package in which the FlightXML3 definitions are stored with a passed in version parameter. While this approach worked, it removed any previous definition from the tcl interpreter and required a complete reload. At FlightAware, HTTP requests are handled by apache with the tcl rivet module. Each child has a tcl interpreter and that interpreter will survive for the duration of the child (typically many requests/responses). If each request is reloading the service definition, then the benefit of a long running interpreter and the optimization of having most packages loaded at creation are lost. Therefore this attempt was abandoned in favor of making changes upstream in tclws.

To add versioning to tclws, a common syntax for defining a version code(s) was required since various tclws methods and definitions will accept version arguments. The syntax for specifying version allows for a single version to be specified, a range of versions or versions above or below some threshold. For example:

version code	valid versions
3	Only valid for version 3
2+	Valid for version 2 and greater
2-	Valid for version 2 or less
2-4	Valid for versions 2 to 4

Since all existing tclws applications will lack any version information, a method or type without version will be considered available and valid for any and all versions.

Unless a tclws service will return a simple type such as an int or char, a custom type is returned. Most FlightXML methods will accept multiple parameters which are expressed as a custom argument type in the tclws service definition. For successful versioning, the data type definitions need to accept version arguments without breaking the existing implementation. Tclws' method ::WS::Utils::ServiceTypeDef is called to create custom data types. When calling that method, the internal dictionary typeInfo is populated with the type's information. An additional optional argument was added to that method to accept a version. Specifying a version for the type will append version information to the dictionary and that type will only be available for valid version calls. In addition to the type itself, we may want to add or remove members of a complex type as our product matures. The type definition consists of a list of field definitions containing fieldName and fieldInfo arguments. FieldInfo consists of key value pairs and version is accepted as a valid key. See below for a service type definition. In the example the TestStruct type is available in versions 2 or greater, the edt field is only available in version 2 (ie removed in version 3) and the reg field is available in versions 3 or greater. The fifth and sixth arguments to ServiceTypeDef are xns namespace and abstract and are left with their default values here.:

```
# Custom type that is only available in version 2 or higher with
# a new member added in version 3
::WS::Utils::ServiceTypeDef Server testService TestStruct {
    ident   {type string comment "flight ident"}
    edt     {type string comment "est. departure time" version
2}
    reg     {type string comment "aircraft tail number" version
3+}
} {} {false} {2+}
```

Methods or services in tclws are the operations performed for a given request. The ::WS::Server::ServiceProc is used for defining those methods, and the second argument, nameInfo, consists of a list of service information. A fourth element for the nameInfo argument is now accepted to indicate an optional version code. If the version is not provided then it will be set to an empty string which is the equivalent of making the method available for all versions. Like type information, tclws stores method definitions in an internal dictionary, procInfo. That dictionary now contains version information for each method. The ServiceProc also defines the method's argument type information. When creating that argument information, the version for that type is set to match the method version and any version data for individual fields will be included. Finally the ServiceProc creates a new proc with the user supplied code body. So that a separate proc need not be defined for every possible version combination of arguments, that proc is defined with a single args argument. Tclws then prepends the user's code block with a call to tcllib's cmdline getKnownOptions to parse those arguments. The possible options are built into a cmdline options list based on the argument type fields with a new argument, tclws_op_version, included. For each argument a local variable is then set matching the original variable name which is available in the user's code. If a tclsh method should be available from version 2 to 4, then the service definition would resemble the following:

```
::WS::Server::ServiceProc testService {WeatherInfo {type
WeatherStruct comment "Returned WeatherInfo results"} {} 2-4}
$arguments $docs {
... # proc definition
}
```

If the service were available since version 2, and a new argument was added in version 3 the definition would be:

```
::WS::Server::ServiceProc testService {WeatherInfo {type
WeatherStruct comment "Returned WeatherInfo results"} {} 2+} {
   ident                 {type string comment "flight ident"
   filedDepartureTime    {type int comment "filed departure
time" version 3+}
} $docs {
... # proc definition
}
```

Since tclws will insert the requested version into the user's code block as the tclws_op_version variable, the user could modify their implementation based on that version:

```
::WS::Server::ServiceProc testService {FlightInfo {type
FlightInfoStruct comment "Returned FlightInfo results"}} {
    ident                {type string comment "flight ident"}
    filedDepartureTime   {type int comment "filed departure time
(epoch value)" version 2+}
} "Example tclws method" {
    if {$tclws_op_version == 3} {
        # look up some data one way
    } else {
        # look up some data another way
    }
}
```

The final component for tclws versioning are changes to how one calls an operation or asks for WSDL/documentation. Each one of those calls now accepts a version argument and will execute the method at the desired version level. Internally, for documentation, the dictionary containing type and method information is filtered based on the requested version before returning the requested documentation to the caller. Again any type or method without version information is considered valid for all versions and calling one of these methods without a version specified will revert to the previous behavior and all type information and methods will be returned. In each case the version argument is passed as "-version {code}" so the relevant calls would be as follows:

```
::WS::Server::callOperation $serviceName $sock -rest -version 3
::WS::Server::generateWsdl $serviceName RivetClient -version 2
::WS::Server::generateInfo $serviceName RivetClient -version 2
::WS::Server::generateJsonInfo $serviceName RivetClient -version
3
```

The version changes to tclws enable FlightXML3 to be extended in the future without redefining existing services or types. As mentioned previously, the FlightXML service is hosted on Apache with the tcl rivet module loaded. The apache configuration captures any version suffix to the FlightXML3 url and appends that version as a parameter to the call:

```
RewriteRule ^/json/FlightXML3.([0-9]+)/([A-Za-z0-9]+)$
/commercial/flightxml/api/rest3.rvt?call=$2&version=$1 [L,QSA]
```

In the apache rivet page (rest3.rvt), the request is checked to see if a version was specified and then subsequent calls for the operation or documentation include that version in the call, as shown above. With these changes FlightXML3 will be able to introduce new methods or modify the return types by incrementing the minor version without impacting customers using a previous version. In particular, customers who use the SOAP interface will not have their clients break when changes to the schema are made, and previous versions’ WSDL will remain unchanged. It is our hope that these changes will be useful to other tclws users and encourage others to experiment with the versioning features and suggest any improvements.