Savon Guide

Savon is a SOAP client library for Ruby. It's goal is to provide a lightweight and easy to use alternative to soap4r. If you're starting to use Savon, please make sure to read this guide and make yourself familiar with SOAP itself, WSDL documents and tools like soapUI.

Table of contents

Installation

Savon is available through Rubygems and can be installed via:

```
$ gem install savon
```

Runtime dependencies

- Builder ~> 2.1.2
- Crack ~> 0.1.8
- HTTPI >= 0.6.0

HTTPI is an interface supporting multiple HTTP libraries. It's a crucial part of Savon and you should make sure to get familiar with it.

Getting started

Savon is based around the Savon::Client object. It represents a particular SOAP service and let's you configure and execute SOAP requests. Let's create a client using a remote WSDL document:

```
client = Savon::Client.new do
  wsdl.document = "http://service.example.com?wsdl"
end
```

`Savon::Client.new` accepts a block to be evaluated in the context of the client object. Inside

this block, you can access all methods from your own class, but local variables won't work. For more information on this, I recommend you read about instance_eval with delegation.

If you don't like this or if it's creating a problem for you, you can use block arguments to specify which objects you would like to receive and Savon will yield those instead of instance evaluating the block. The `.new` method accepts 1-3 arguments and yields the following objects:

```
[wsdl, http, wsse]
```

For example, to work with the wsdl and http object, you can specify only two of the three possible arguments:

```
client = Savon::Client.new do |wsdl, http|
  wsdl.document = "http://service.example.com?wsdl"
  http.proxy = "http://proxy.example.com"
end
```

The three objects mentioned above can also be used after instantiating the client (outside of the block). For example:

```
client.wsse.credentials "username", "password"
```

The next sections should give you a pretty good impression on how these objects can be used.

The WSDL object

The wsdl object is actually called Savon::WSDL::Document, but I'll refer to these objects by shortnames. The wsdl object is a representation of a WSDL document.

Inspecting a Service

Specifying the location of a WSDL document gives you access to a couple of methods for inspecting your service.

```
# specifies a remote location
wsdl.document = "http://service.example.com?wsdl"

# uses a local document
wsdl.document = "../wsdl/authentication.xml"
```

The following examples assume you specified a WSDL location.

```
# returns the target namespace
wsdl.namespace  # => "http://v1.example.com"

# returns the SOAP endpoint
wsdl.endpoint  # => "http://service.example.com"

# returns an Array of available SOAP actions
wsdl.soap_actions  # => [:create_user, :get_user, :get_all_users]

# returns the WSDL document as a String
wsdl.to_xml  # => "<wsdl:definitions name=\"AuthenticationService\" ..."
```

Note: your service probably uses (lower)CamelCase method and object names, but Savon maps those to snake_case Symbols for you.

Working without a WSDL

Retrieving and parsing WSDL documents is a quite expensive operation. And even though Savon caches the result, my recommendation is to not use a WSDL document (at least in production) and directly access the SOAP endpoint instead. This requires you to specify the SOAP endpoint and target namespace instead of a WSDL location:

```
client = Savon::Client.new do
  wsdl.endpoint = "http://service.example.com"
  wsdl.namespace = "http://v1.example.com"
end
```

The HTTP object

HTTPI::Request is provided by the HTTPI gem and represents an HTTP request. Savon executes a GET request to retrieve remote WSDL documents and POST requests for each SOAP request.

I'm only going to document a few interesting details and point you to the HTTPI documentation for additional information.

Note: HTTPI is still a very young project and might not support everything you need. Please don't hesitate to file bugs or make wishes for the library to support additional features.

SSL and HTTP headers

You can easily set the SSL certificate and custom HTTP headers for your requests:

```
client = Savon::Client.new do
  http.auth.ssl.cert_key_file = 'cert.key'
  http.auth.ssl.cert_key_password = 'C3rtP@ssw0rd'
  http.auth.ssl.cert_file = 'cert.crt'
  http.auth.ssl.verify_mode = :none
  http.read_timeout = 90
  http.open_timeout = 90
  http.headers = { "Accept-Encoding" => "gzip, deflate", "Connection" => "Keep-Alive" }
end
```

Headers can also be changed on subsequent calls of the client.request using the same approach.

SOAPAction

SOAPAction is an HTTP header information required by legacy services. If present, the header value must have double quotes surrounding the URI-reference (SOAP 1.1. spec, section 6.1.1). Here's how you would set/overwrite the SOAPAction header:

```
http.headers["SOAPAction"] = '"urn:example#service"'
```

Cookies

If your service relies on cookies to handle sessions, you can grab the cookie from the HTTPI::Response and set it for the next request:

```
client.http.headers["Cookie"] = response.http.headers["Set-Cookie"]
```

The WSSE object

Savon::WSSE allows you to use WSSE authentication (PDF).

```
# sets the WSSE credentials
wsse.credentials "username", "password"

# enables WSSE digest authentication
wsse.credentials "username", "password", :digest
```

Executing SOAP requests

Now for the fun part. To execute SOAP requests, `Savon::Client#request` is the way to go. Let's look at a very basic example of executing a SOAP request to a `get_all_users` action.

```
response = client.request :get_all_users
```

This single argument (the name of the SOAP action to call) works in different ways depending on whether you specified a WSDL document to use. If you did, Savon will parse the WSDL document for available SOAP actions and convert their names to snake_case Symbols for you. When you're not using a WSDL, the argument will (by convention) be converted to lowerCamelCase.

```
:get_all_users.to_s.lower_camelcase  # => "getAllUsers"
:get_pdf.to_s.lower_camelcase        # => "getPdf"
```

This convention might not work for you if your service requires CamelCase method names or methods with UPPERCASE acronyms. But don't worry. If you pass in a String instead of a Symbol, Savon will not convert the argument.

```
response = client.request "GetPDF"
```

The argument(s) passed to the `#request` method will affect the SOAP input tag inside the SOAP request. To make sure you know what this means, here's an example for a simple request:

```
<env:Envelope
    xmlns:env="http://schemas.xmlsoap.org/soap/envelope/"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <env:Body>
    <getAllUsers />  <!-- the SOAP input tag -->
  </env:Body>
</env:Envelope>
```

By now you should know the result of passing a single argument. But fairly often you need to prefix the input tag with the target namespace of your service like this:

```
<env:Envelope
    xmlns:env="http://schemas.xmlsoap.org/soap/envelope/"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema"
```

```
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:wsdl="http://v1.example.com">
  <env:Body>
    <wsdl:getAllUsers />
  </env:Body>
</env:Envelope>
```

If you pass two arguments to the `#request` method, the first (a Symbol) will be used for the namespace and the second (a Symbol or String) will be the SOAP action to call:

```
response = client.request :wsdl, :get_all_users
```

On rare occasions, you may actually need to attach XML attributes to the input tag. In that case, you can pass a Hash of attributes to the name of your SOAP action and the optional namespace:

```
response = client.request :wsdl, "GetPDF", :id => 1
```

These three arguments will generate the following input tag:

```
<wsdl:GetPDF id="1" />
```

Since most SOAP actions require you to pass arguments for e.g. the user to return, you need to send a “payload”. Luckily you’re already familiar with passing a block to a method, right? `Savon::Client#request` also accepts a block for you to access the following objects:

```
[soap, wsdl, http, wsse]
```

Notice, that the list is almost the same as the one for `Savon::Client.new`. Except now, there is an additional object called soap. In contrast to the other three objects, the soap object is tied to single requests. Savon creates a new soap object for every request.

The SOAP object

Savon::SOAP::XML is tied to a single SOAP request and lets you customize the SOAP request XML.

SOAP version

Savon by default expects your services to be based on SOAP 1.1. For SOAP 1.2 services, you can set the SOAP version per request:

```
response = client.request :get_user do
  soap.version = 2
end
```

Namespaces

If you don’t pass a namespace to `Savon::Client#request`, Savon will register the target namespace (“xmlns:wsdl”) for you. If you did pass a namespace, Savon will use it instead of the default one. For example:

```
client.request :v1, :get_user
```

```
<env:Envelope
    xmlns:env="http://schemas.xmlsoap.org/soap/envelope/"
```

```
    xmlns:xsd="http://www.w3.org/2001/XMLSchema"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:v1="http://v1.example.com">
  <env:Body>
    <v1:GetUser>
  </env:Body>
</env:Envelope>
```

You can always set namespaces or overwrite namespaces set by Savon. Namespaces are stored as a simple Hash.

```
# setting a new namespace
soap.namespaces["xmlns:g2"] = "http://g2.example.com"

# overwriting the "xmlns:wsdl" namespace
soap.namespaces["xmlns:wsdl"] = "http://ns.example.com"
```

SOAP body

You probably need to specify some arguments required by the SOAP action you're going to call. If you're, for example, interacting with a `get_user` action which expects the ID of the user to return, you can simply pass a Hash:

```
response = client.request :get_user do
  soap.body = { :id => 1 }
end
```

As you already saw before, Savon is based on a few conventions to make the experience of having to work with SOAP and XML as pleasant as possible. The Hash passed to `Savon::SOAP::XML#body=` is not an exception. It is translated to XML using the `Hash#to_soap_xml` method provided by Savon.

Here's a more complex example:

```
response = client.request :wsdl, "CreateUser" do
  soap.body = {
    :first_name => "The",
    :last_name  => "Hoff",
    "FAME"      => ["Knight Rider", "Baywatch"]
  }
end
```

As with the SOAP action, Symbol keys will be converted to lowerCamelCase and String keys won't be touched. The previous example generates the following XML:

```
<env:Envelope
    xmlns:env="http://schemas.xmlsoap.org/soap/envelope/"
    xmlns:wsdl="http://v1.example.com">
  <env:Body>
    <wsdl:GetUser>
      <firstName>The</firstName>
      <lastName>Hoff</lastName>
      <FAME>Knight Rider</FAME>
      <FAME>Baywatch</FAME>
    </wsdl:GetUser>
  </env:Body>
</env:Envelope>
```

Some services actually require the XML elements to be in a specific order. If you don't use

Ruby 1.9 (and you should), you can not be sure about the order of Hash elements and have to specify the correct order using an Array under a special `:order!` key:

```
{ :last_name => "Hoff", :first_name => "The", :order! => [:first_name, :last_name] }
```

This will make sure, that the lastName tag follows the firstName.

Assigning arguments to XML tags using a Hash is even more difficult. It requires another Hash under an `attributes!` key containing a key matching the XML tag and the Hash of attributes to add:

```
{ :first_name => "TheHoff", :last_name => nil, :attributes! => { :last_name => { "xsi:nil" => true } } }
```

This example will be translated to the following XML:

```
<firstName>TheHoff</firstName><lastName xsi:nil="true"></lastName>
```

I would not recommend using a Hash for the SOAP body if you need to create complex XML structures, because there are better alternatives. One of them is to pass a block to the `Savon::SOAP::XML#body` method. Savon will then yield a `Builder::XmlMarkup` instance for you to use.

```
soap.body do |xml|
  xml.firstName("The")
  xml.lastName("Hoff")
end
```

Last but not least, you can also create and use a simple String (created with Builder or any another tool):

```
soap.body = "<firstName>The</firstName><lastName>Hoff</lastName>"
```

SOAP header

Besides the body element, SOAP requests can also contain a header with additional information. Savon sees this header as just another Hash following the same conventions as the SOAP body Hash.

```
soap.header = { "SecretKey" => "secret" }
```

Custom XML

If you’re sure that none of these options work for you, you can completely customize the XML to be used for the SOAP request:

```
soap.xml = "<custom><soap>request</soap></custom>"
```

The Response object

`Savon::Client#request` returns a Savon::SOAP::Response for you to work with. While `Savon::SOAP::Response#to_hash` converts the SOAP response XML to a Ruby Hash:

```
response.to_hash  # => { :response => { :success => true, :name => "John" } }
```

`Savon::SOAP::Response#to_xml` simply returns the original SOAP response XML:

```
response.to_xml  # => "<response><success>true</success><name>John</name></response>"
```

The response also contains the HTTPI::Response:

```
response.http  # => #<HTTPI::Response:0x1017b4268 ...
```

Error handling

By default, Savon raises both `Savon::SOAP::Fault` and `Savon::HTTP::Error` when encountering these kind of errors.

```
begin
  client.request :get_all_users
rescue Savon::SOAP::Fault => fault
  log fault.to_s
end
```

Both errors inherit from `Savon::Error`, so you don't need to explicitly rescue both:

```
begin
  client.request :get_all_users
rescue Savon::Error => error
  log error.to_s
end
```

If you changed the default to not raise these errors, you can ask the response whether the request was successful:

```
response.success?     # => false
response.soap_fault?  # => true
response.http_error?  # => false
```

You can then access the error objects mentioned above:

```
response.soap_fault  # => Savon::SOAP::Fault
response.http_error  # => Savon::HTTP::Error
```

Please notice, that these methods always return an error object. To check if an error is actually present, you can either ask the response or directly ask the error object:

```
response.soap_fault.present?  # => true
response.http_error.present?  # => false
```

Global configuration

Logging

By default, Savon logs each SOAP request and response to STDOUT using a log level of :debug.

```
Savon.configure do |config|
  config.log = false            # disable logging
  config.log_level = :info      # changing the log level
  config.logger = Rails.logger  # using the Rails logger
end
```

Error handling

If you don’t like to rescue errors, here’s how you can tell Savon to not raise them:

```
Savon.configure do |config|
  config.raise_errors = false  # do not raise SOAP faults and HTTP errors
end
```

SOAP version

Also changing the default SOAP version of 1.1 to 1.2 is fairly easy:

```
Savon.configure do |config|
  config.soap_version = 2  # use SOAP 1.2
end
```

Ecosystem

Savon::Model

Savon::Model creates SOAP service oriented models.

Savon::Spec

Savon::Spec helps you test your SOAP requests.

Alternative libraries

And if you feel like there’s no way Savon will fit your needs, you should take a look at The Ruby Toolbox to find some alternatives.