Tool Demonstration: Testing JSON Web Services Using jsongen

Ignacio Ballesteros
Luis Eduardo Bueso
Lars-Åke Fredlund
Julio Mariño

September 18, 2018

UPM

Contents

Context

What is jsongen

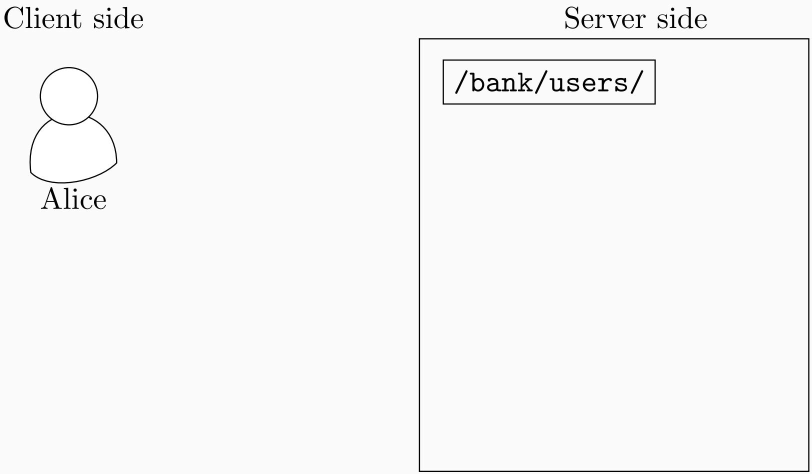
A tool for testing web services based on json communication.

Using the Quviq's Quichckeck state machine, jsongen can test the dynamic properties of web services.

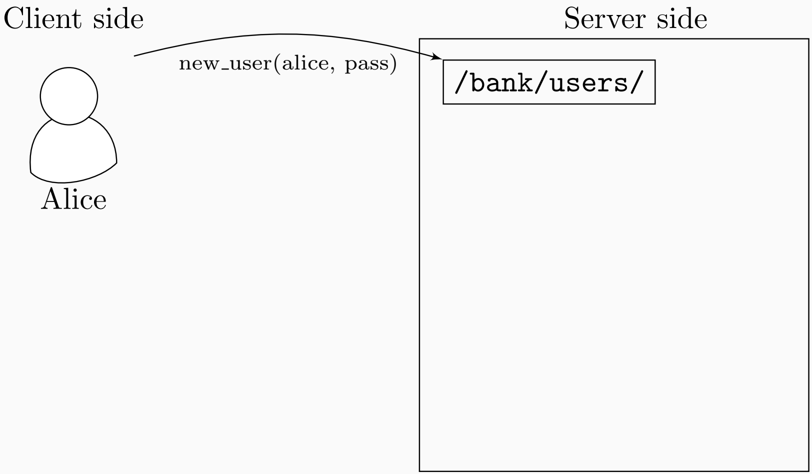
We can start testing only with a JSON Schema file defining an API.

This makes web services testing easier and faster.

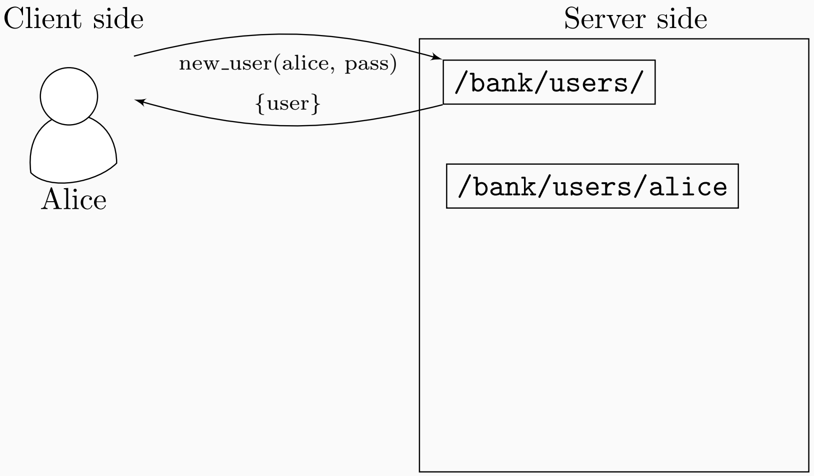
Our bank web service



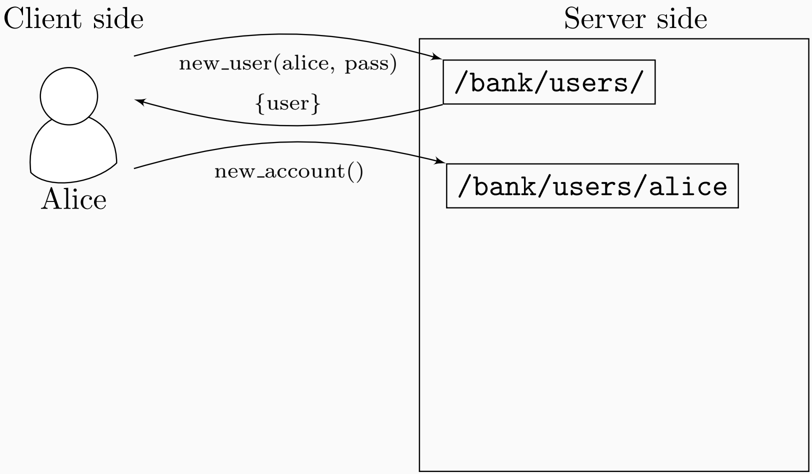
Our bank web service



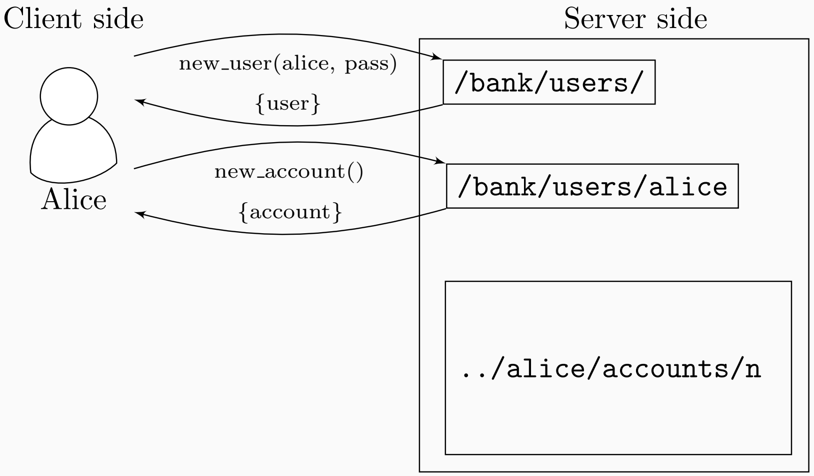
Our bank web service



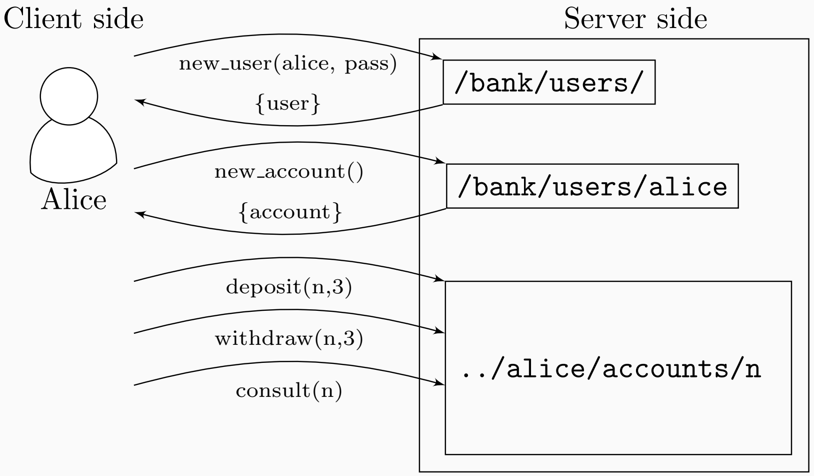
Our bank web service



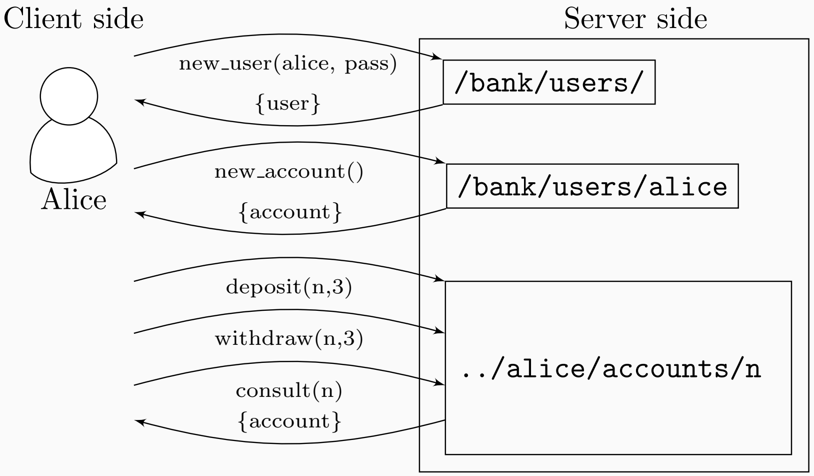
Our bank web service



Our bank web service



Our bank web service



Testing a web service operation using jsongen

Objectives and example

The main objective of this example is to give a general idea of how to use jsongen to test a simple web service operation.

The web service operation:

Operation	new user
URI	http://localhost:5000/bank/users/
Method	POST
Body	name: string, password: string
Result	user: string
Status	201

Starting out our JSON Schema

Operation	new user
URI	http://localhost:5000/bank/users/
Method	POST

```
{
  "rel": "new_user",
  "href": "http://localhost:5000/bank/users/",
  "title": "new user",
  "method": "POST",
  ...
```

Body generator

Body	name: string, password: string

```
...
"schema": {
  "type": "object",
  "required": ["user",
               "password"],
  "properties": {
    "user": {
      "type": "string"
    },
    "password": {
      "type":  "string"
    }
  },
  "additionalProperties":
  false
}
```

Body generator

Body	name: string, password: string

```
...
"schema": {
  "type": "object",
  "required": ["user",
               "password"],
  "properties": {
    "user": {
      "type": "string"
    },
    "password": {
      "type":  "string"
    }
  },
  "additionalProperties":
  false
}
```

```
{
  "user": "sxa2",
  "password": "vxkj"
}
```

Body generator: self-defined generators

Body	name: string, password: string

```
...
"schema": {
  "type": "object",
  "required": ["user", "password"],
  "properties": {
    "user": {
      "quickcheck": { "name": "bank_generators:gen_user" }
    },
    "password": {
      "quickcheck": { "name": "bank_generators:gen_password" }
    }
  },
  "additionalProperties": false
...
```

Response validation

Result	user: string
Status	201

```
{
  "type": "object",
  "required": ["user"],
  "status": 201,
  "properties": {
    "user": {
      "type":
      "string"
    }
  },
  "additionalProperties":
  false
}
```

Response validation

Result	user: string
Status	201

```
{
  "type": "object",
  "required": ["user"],
  "status": 201,
  "properties": {
    "user": {
      "type":
      "string"
    }
  },
  "additionalProperties":
  false
}
```

```
{
  "user": "sxa2"
}
```

Structure

At the end we will have 2 files:

- `new_user.jsch` which contains the information used in the request generation.
- `new_user_response.jsch` which contains the information in the response validation.

The last important JSON Schema identifier is:

```
...
"targetSchema": {
  "$ref": "new_user_response.jsch#"
}
...
```

Demo

Testing a web service dynamic state using jsongen

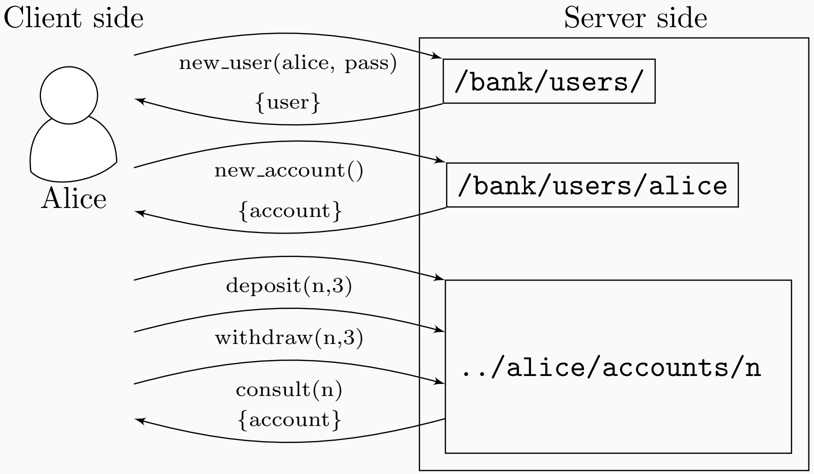
Objectives and API description

The main objective of this example is to give a general idea of how to use the dynamic links descovering habilities of jsongen.

In this example we will test the protocol of the whole bank API.

Let's revisit our possible operations:

Bank api operations



Dynamic discovery of operations

Jsongen can create sequences of operations with data received in previous requests.

When jsogen validates a response, we can define a new link to explore within the JSON Schema.

Our `create_account` operation unlocks three operations over the account created:

- consult
- deposit
- withdraw

Operation: new account

Operation	new account
URI	http://localhost:5000/bank/users/{user}/accounts/
Method	POST
Body	empty

```
{
  "rel": "new_account",
  "href": "http://localhost:5000/bank/users/{user}/accounts/",
  "title": "new account",
  "method": "POST",
  "schema": {
    "type": "object",
    "additionalProperties": false,
    "properties": {}
  }
}
```

Operation: new account

Result	accountid: string, balance: integer, owner: string
Status	201

```
{
  "type": "object",
  "required": ["accountid", "balance", "owner"],
  "status": 201,
  "properties": {
    "accountid": { "type": "string" },
    "balance": { "type": "integer" },
    "owner": { "type": "string" }
  },
  "additionalProperties": false,
}
```

Dependencies

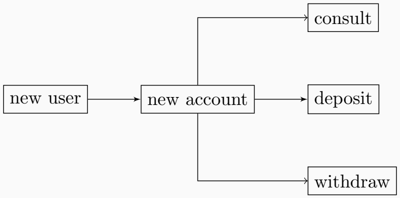


Figure 1: Operation availability dependency

Structure

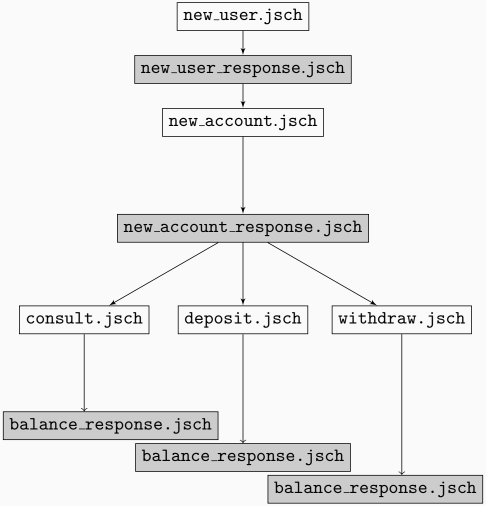


Figure 2: File discovery structure

Demo

Testing a web service state correctness with a jsongen model

Objectives and example

The main objective of this example is to give a general idea of how to use jsongen to test the state of a web service.

The web service state:

Operation	**Changes the state**
new user	yes
new account	yes
consult account	no
withdraw	yes
deposit	yes

The model interface

To use the model we need to implement the next 3 functions in an erlang module:

```
-export([initial_state/0, next_state/4, postcondition/4]).

initial_state() ->
  ...

next_state(Super, State, Result, Call) ->
  ...

postcondition(Super, State, Call, Result) ->
  ...
```

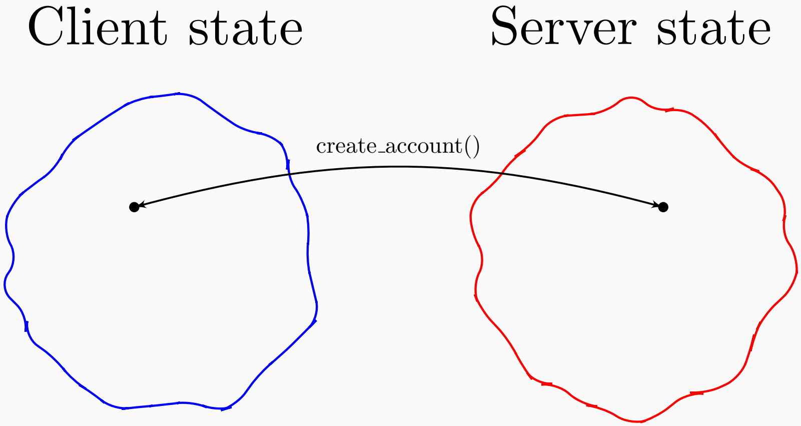
State

We will model our state as:

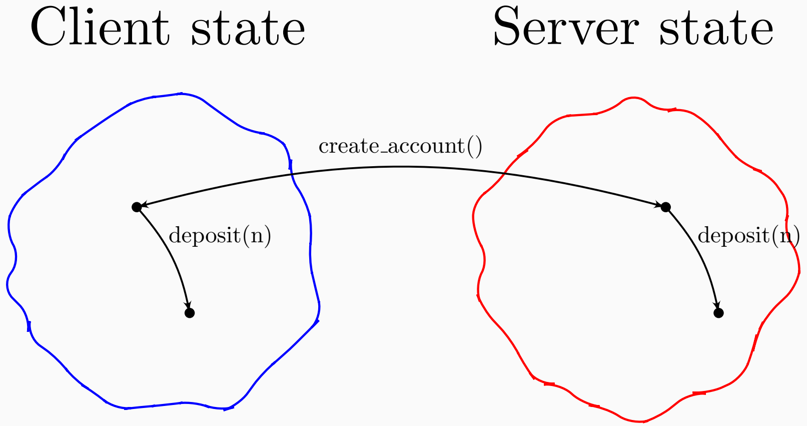
```
-record(state, {users, accounts}).

initial_state() ->
  #state
    {
     users = [],
     accounts = #{}
    }.
```

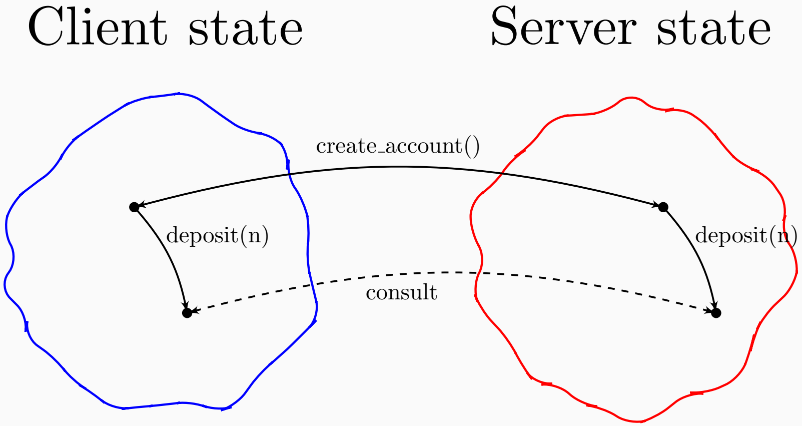
State analysis



State analysis



State analysis



Model: next_state

This function changes our model' state.

```
next_state(Super, State, Result, Call) ->
  Info = get_info(Call, State, Result),
  NextModelState = next_model_state(Info#info.op_title,
                                    Info#info.priv_state,
                                    Info#info.call_body,
                                    Info#info.json_res),
  NewState = jsg_links_utils:
    set_private_state(NextModelState, State),
  Super(NewState, Result, Call).
```

Model: next_model_state

```
next_model_state(Operation, ModelState, {struct,BodyValues},
                 {struct,Values}) ->
  case Operation of
    "new_user" ->
      case proplists:lookup(<<"user">>, Values) of
        {_, User} ->
          ModelState#state {
            users = [User|ModelState#state.users]
           };
        none -> ModelState
      end;

    ...
```

```
...
"new account" ->
   case {proplists:lookup(<<"accountid">>, Values),
         proplists:lookup(<<"balance">>, Values)} of
     {{_, AccountId}, {_, Balance}} ->
       ModelState#state {
         accounts = maps:put(AccountId,
                             Balance,
                             ModelState#state.accounts)
        };
     _ -> ModelState
   end;
 ...
```

Model: postcondition_model_state

```
postcondition_model_state(Operation, ModelState,
                          {struct, Values}) ->
  maps:keys(maps:filter(fun(AccountId, Balance) ->
                              Balance < 0
                            end,
                            ModelState#state.accounts)) == []
    and case Operation of
          "consult account" ->
            case {proplists:lookup(<<"accountid">>, Values),
                  proplists:lookup(<<"balance">>, Values)} of
              {{_, AccountId}, {_, Balance}} ->
                Balance == maps:get(AccountId, ModelState#state.
              _ -> false
            end;
          _ -> true
        end.
```

Demo

Conclusion

Conclusion

What jsongen does:

- Automatic test case generation.
- Trazable errors.
- Extensible library to model service state.
- Property-based testing of web services.

What jsongen needs:

- A JSON Schema specification of the API.
- No programming knowledge needed for basic usage.
- Erlang knowledge for advanced usage.