Tool Demonstration: Testing JSON Web Services Using jsongen

Ignacio Ballesteros
Luis Eduardo Bueso
Lars-Åke Fredlund
Julio Mariño
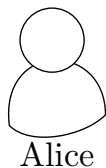
September 19, 2018

UNIVERSIDAD
POLITÉCNICA
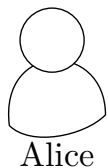DE MADRID

POLITÉCNICA

Our bank web service

Client side

Alice

Server side

/bank/users/

POLITÉCNICA

Our bank web service

Client side

Alice

new_user(alice, pass)

Server side

/bank/users/

Our bank web service
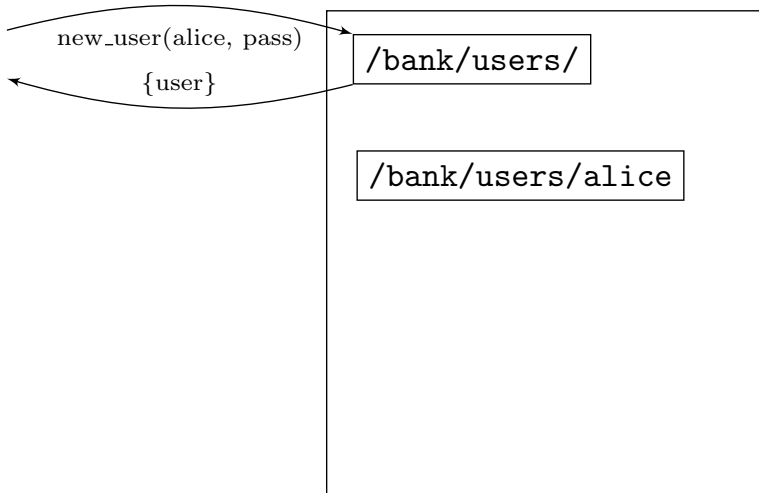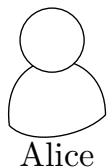
Client side

Alice

new_user(alice, pass)

{user}

Server side

/bank/users/

/bank/users/alice

POLITÉCNICA

Our bank web service

Client side   Server side
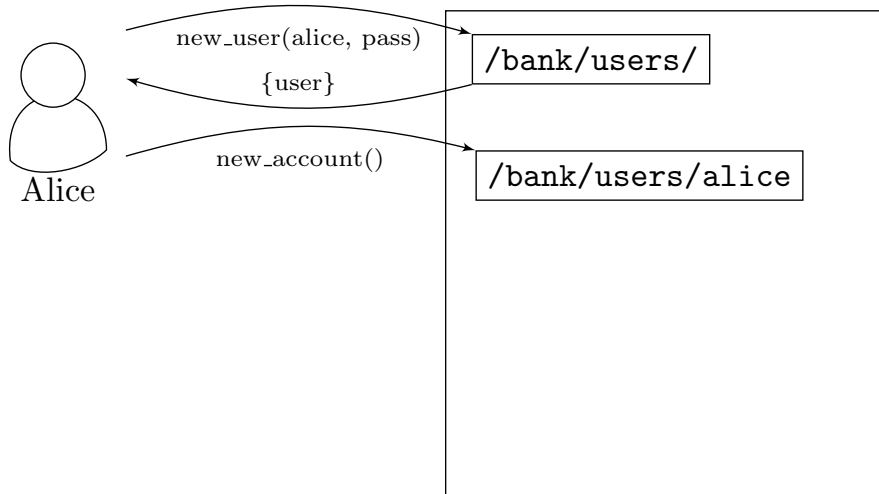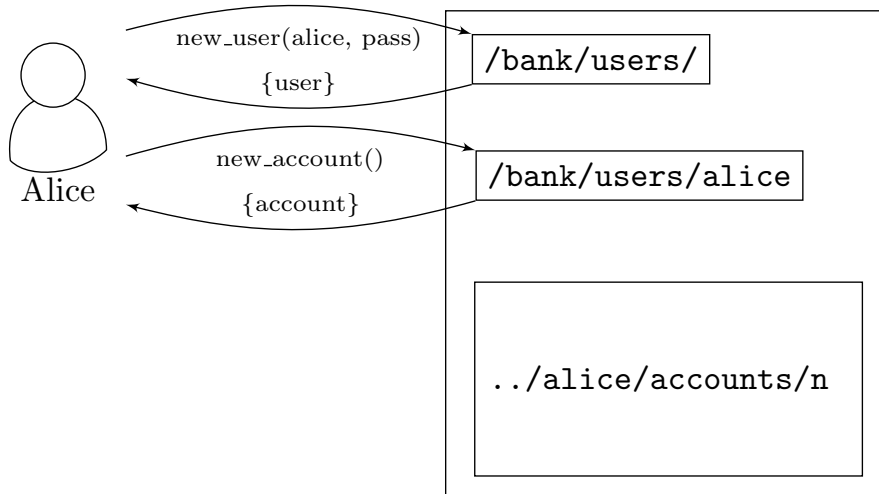
new_user(alice, pass)

{user}

/bank/users/

new_account()

/bank/users/alice

Alice

POLITÉCNICA

Our bank web service

POLITÉCNICA

Our bank web service

POLITÉCNICA
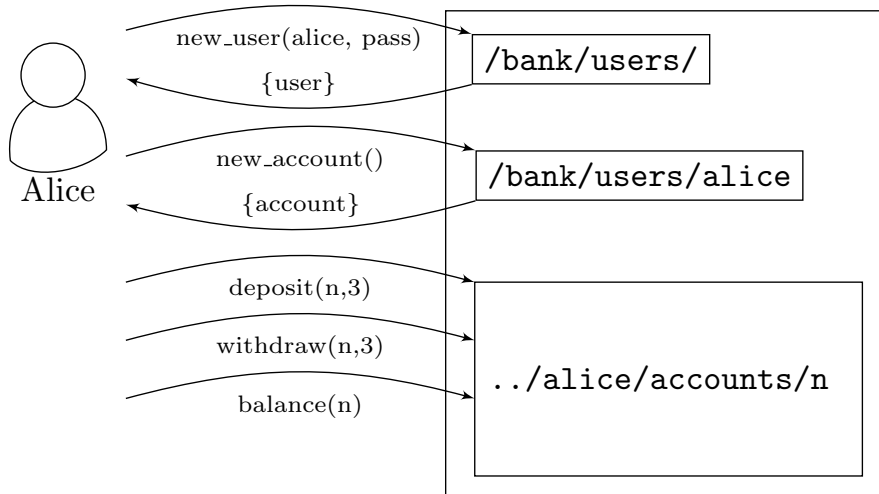
Our bank web service

What is jsongen

- Jsongen is a tool for testing web services based on JSON communication.
- We can generate automated and random test cases using Quickcheck.
- What do we need:
  - A JSON Schema for the tested API.
  - Optionally, an Erlang module for state checking.
- Differences with other testing tools:
  - Automated test cases.

POLITÉCNICA

Testing a single web service operation using jsongen
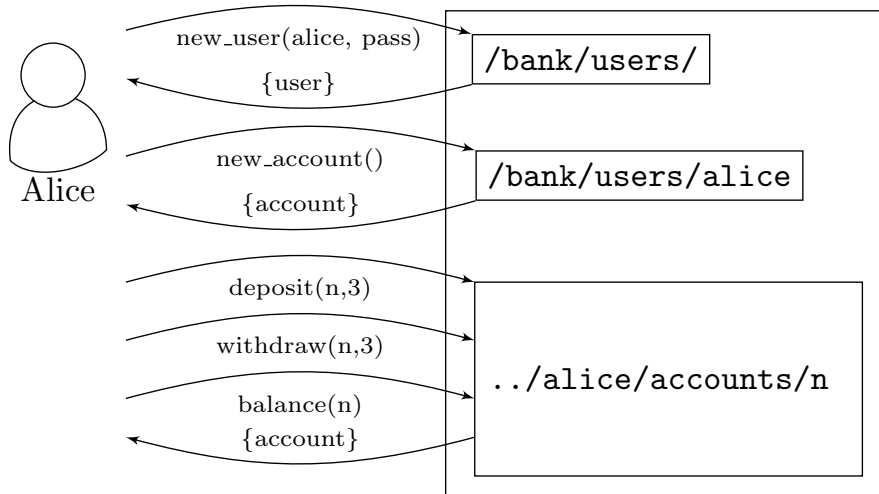
Operation: new user

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

POLITÉCNICA

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",
  ...
```

Automatic body generation

Creating the new user body

Body	name: string, password: string

JSON Schema definition   JSON generated

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

POLITÉCNICA

Automatic body generation

Creating the new user body

Body	name: string, password: string

JSON Schema definition

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

JSON generated

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

POLITÉCNICA

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

Validating the new user response

Result	user: string
Status	201

JSON Schema definition   Valid JSON

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

POLITÉCNICA

Response validation

Validating the new user response

Result	user: string
Status	201

JSON Schema definition

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

Valid JSON

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

POLITÉCNICA

JSON Schema files relationships

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#"
}
...
```

POLITÉCNICA

Demo

POLITÉCNICA

Bank api operations

Client side   Server side

Alice

new_user(alice, pass)   /bank/users/
{user}

new_account()   /bank/users/alice
{account}

deposit(n,3)

withdraw(n,3)   ../alice/accounts/n

balance(n)
{account}

POLITÉCNICA
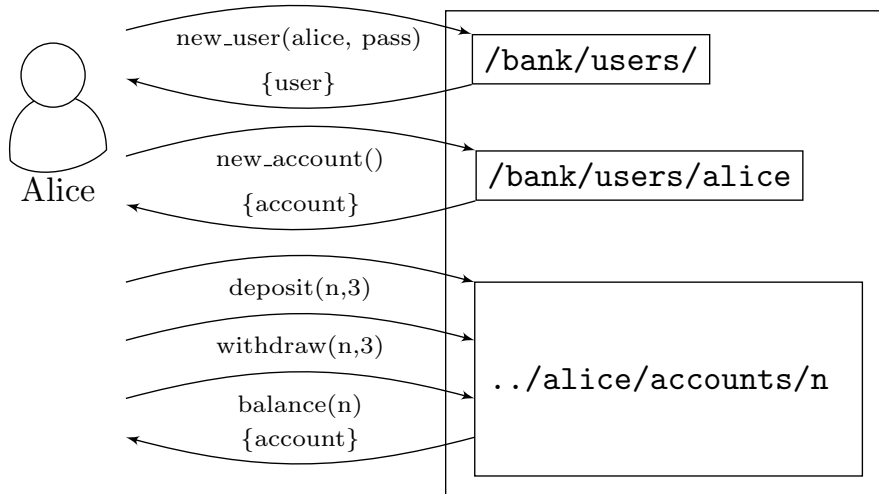
Objectives and API description

- 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:

Operation	Resource identifiers
new user	/bank/users/
new account	/bank/users/{user}/accounts/
consult account	/bank/users/{owner}/accounts/{accountid}/
deposit	/bank/users/{owner}/accounts/{accountid}/
withdraw	/bank/users/{owner}/accounts/{accountid}/

POLITÉCNICA

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:
  - ▶ balance
  - ▶ deposit
  - ▶ withdraw

POLITÉCNICA

Operation: new account

We need a user in order to create a new account. This user is taken from the new_user response:

```
{ "user": "alice" }
```

Operation: new account

We need a user in order to create a new account. This user is taken from the
new_user response:

```
{ "user": "alice" }
```

We create our next request with a reference to the user value returned:

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

POLITÉCNICA

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,
}
```

Operation: new account

Now we define the operations unlocked when we create an account.

```
{
  "links": [
    {
      "title": "account balance",
      "method": "GET",
      "href": ".../bank/users/{owner}/accounts/{accountid}/",
      "rel": "balance",
      "targetSchema": {
        "$ref": "balance_account_response.jsch#"
      }
    },
    { "title": "deposit", ... },
    { "title": "withdraw", ... }
  ]
}
```

POLITÉCNICA

Operation availability dependency

POLITÉCNICA

Demo

POLITÉCNICA
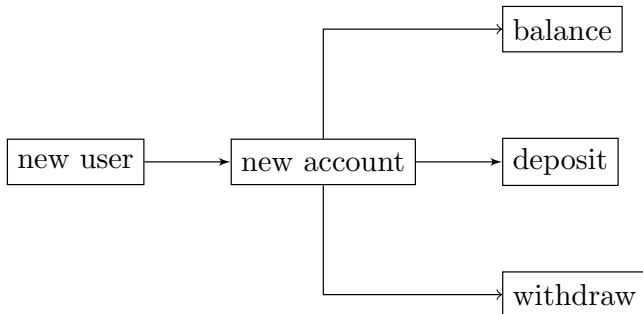
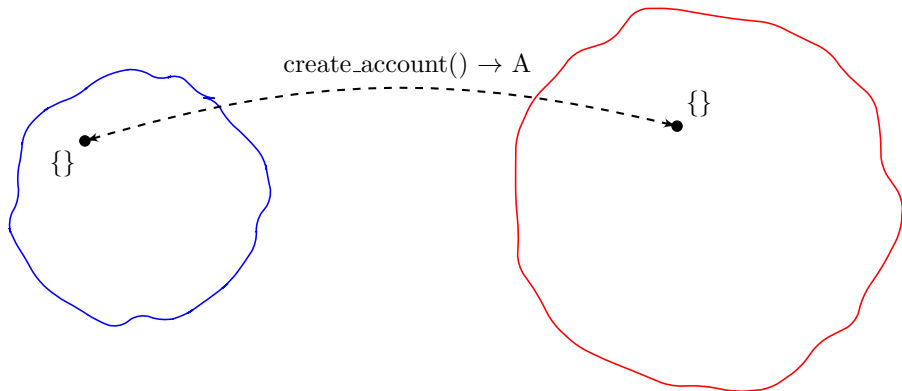# Testing a web service state correctness with a jsongen model

- 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
balance	no
withdraw	yes
deposit	yes

POLITÉCNICA

State analysis

Abstract state machine

Server state

$$\text{create\_account}() \rightarrow \text{A}$$

$\{\}$

$\{\}$

State analysis

Abstract state machine

Server state

$$\mathrm{create\_account}() \rightarrow A$$

{} A {A:0}

{} A {A:0}

POLITÉCNICA

State analysis

Abstract state machine

Server state

$\text{create\_account}() \rightarrow \text{A}$

$\text{deposit}(\text{A},10)$

{}

A

{A:0}

{}

A

{A:0}

POLITÉCNICA

State analysis

Abstract state machine

Server state

$create\_account() \rightarrow A$

$deposit(A,10)$

$\{\}$ A $\{A:0\}$ $A+=10$ $\{A:10\}$

$\{\}$ A $\{A:0\}$ $A+=10$ $\{A:10\}$

POLITÉCNICA
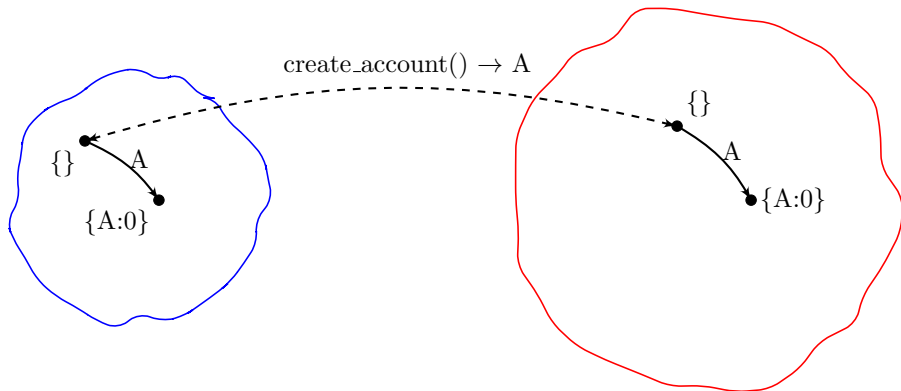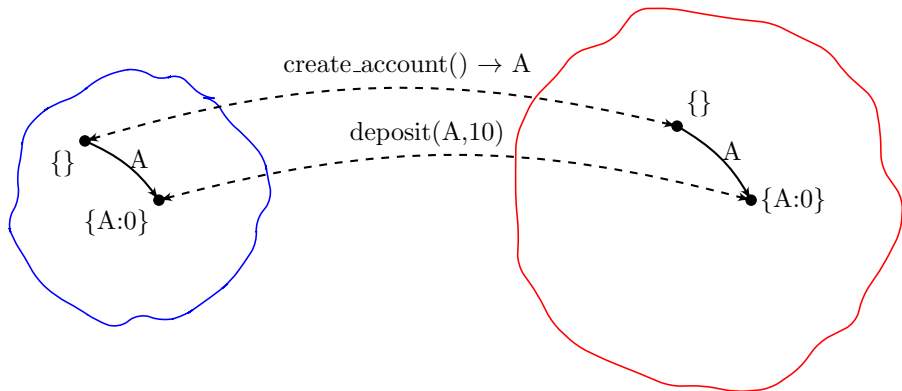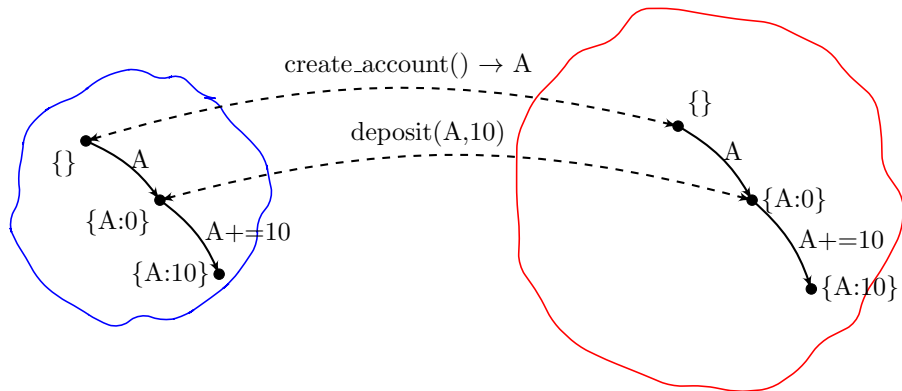
Abstract state machine interface

To program the abstract state machine we need to implement the following functions:

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

initial_state() ->
  ...

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

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

POLITÉCNICA

Abstract state machine bank intial state

We will model our state as:

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

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

POLITÉCNICA

Abstract state machine bank next_state

```erlang
next_state(Super, State, Result, Call) ->
  ...
  case Operation of
    "new_user" ->
      case proplists:lookup(<<"user">>, Values) of
        {_, User} ->
          ModelState#state {
            users = [User|ModelState#state.users]
          };
        none -> ModelState
      end;
    ...
```

```erlang
...
"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;
...
```

Abstract state machine bank postcondition_state

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

POLITÉCNICA

Demo

POLITÉCNICA

Summary

What jsongen does:

- Automatic test case generation.
- Traceable 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.

Jsongen is a public tool available at:

```
https://github.com/fredlund/jsongen
```

POLITÉCNICA