Mocker

API : Api is the one that you have to mock. It has the following parameters

- method – String – Possible Values (case sensitive) **GET**, **POST**, **HEAD**, **OPTIONS**, **PATCH**, **PUT**, **DELETE** and **Any**. “Any” can be used to specify that the api can be available for any types of methods
- route – String – The path of the api. A prefix(/api/v1/mock) will be added to this route while accessing this api
- status – integer – This is the response status number the api has to return.
- response_headers – map[string]string – The default headers for the response
- response_body – string – This is a go template and will be parsed and executed with the variable. See available variables below. This can evaluate to a html or json or anything. Remember to change the header content type accordingly

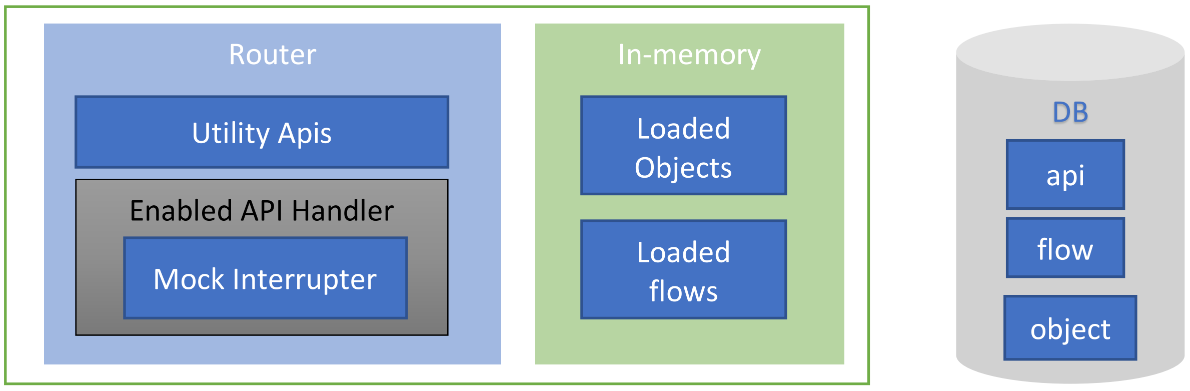
PS: Please make sure that different apis don’t have conflicting Paths. Example: /something/user/anything and /something/:variable/anything are conflicting as the router cannot resolve the word ‘user’ with the variable.

Object: This is a plain JavaScript object hereafter referred to as JsonObject. Stored Objects(in DB) is referred by their auto generated Id.

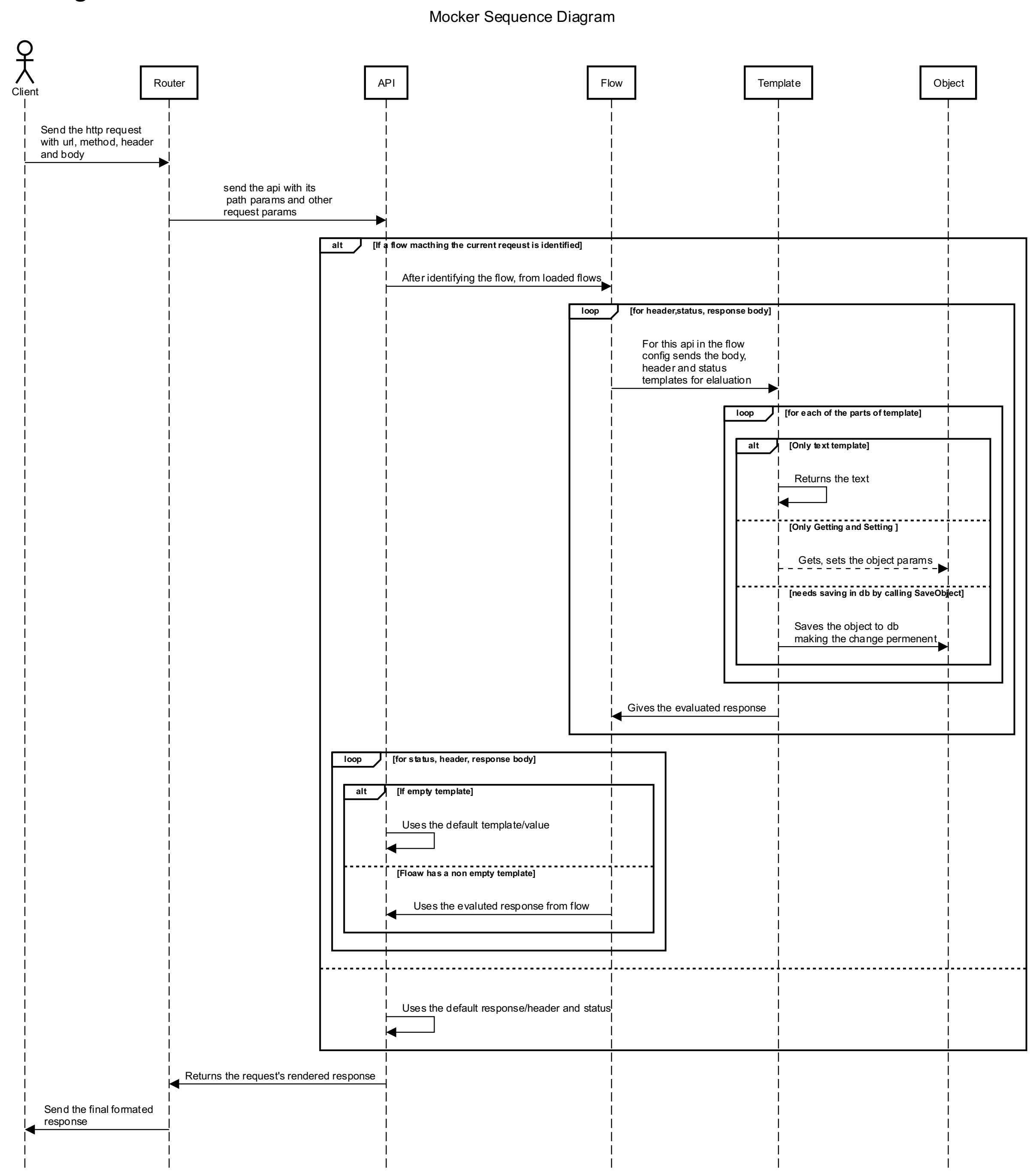
Flow: A flow is a single test case scenario. A flow can have one or more api calls and depending on the scenario these APIs can give different responses. A flow is identified and loaded using its auto generated **Id**. So a flow have the following

- Title – string -
- Objects – List of Ids(of object) – This is used to maintain the states across multiple api calls. The objects mentioned in this list will be pre-loaded in memory for eas of usage
- Identifier – string – This is a go template that would evaluate to “true” for this flow to be used in a scenario. Among the loaded flows, for each of the requests the identifier will be executed to identify which flow has to be used for a request. If multiple flows satisfy and evaluate to true then the 1st flow that was loaded would be used. The template can only use objects that are enabled and are mentioned in the Object list to be used. Else the objects might not get evaluated.
- Config – map[string]{status, header, responseBody} – The config has a map for each of the api it needs to be part of it. A api is defined by its method and route(excluding the prefix. Example: “GET /test/:exactVariable/api” . The method should be same as the one defined in the API list. In other words, if the method is defined as Any then GET or POST cannot be used in the config. The Status, Header and the ResponseBody are all go templates. If any of these are empty then the default for that api would be used.

Architecture Diagram



Sequence Diagram:


Mocker Sequence Diagram
Client
Router
API
Flow
Template
Object
Send the http request with url, method, header and body
send the api with its path params and other request params
alt
[If a flow macthing the current reqeust is identified]
After identifying the flow, from loaded flows
loop
[for header,status, response body]
For this api in the flow config sends the body, header and status templates for elaluation
loop
[for each of the parts of template]
alt
[Only text template]
Returns the text
[Only Getting and Setting]
Gets, sets the object params
[needs saving in db by calling SaveObject]
Saves the object to db making the change permenent
Gives the evaluated response
loop
[for status, header, response body]
alt
[If empty template]
Uses the default template/value
[Floaw has a non empty template]
Uses the evaluted response from flow
Uses the default response/header and status
Returns the request's rendered response
Send the final formated response

Template used : Golang html/template (the html template is implemented on top of text template). Some of the useful read is available at the end of this document
Refer:

- https://golang.org/pkg/text/template/
- https://golang.org/pkg/html/template/

Variables Available:

SNo	Variable	Explanation	Example
1	Method Type: string	This is the request method like “GET”, “POST”,”PUT” etc	{{if eq .Method “GET”}}This is a Get Request{{end}}
2	Header Type: map[string][]string	This has the request headers. For each of the header value, there can be multiple values if the requests sets multiple values. So its []string instead of string	index .Header.Accept 1 PS: In templates the index starts from 1 (unlike in most languages)
3	JSON Type: Json Object	This is the request JSON if the request’s content type is application/json	Get .JSON “user” “first_name” Refer the Get function below
4	PathParams Type: map[string]string	This the list of path params of the api. For example the api is https://../KISSHT/:accountId/loan then one of the path param would be accountId	.PathParam.accountId
5	Query Type: map[string][]string	This is the get params that are part of the url	
6	Form Type: map[string][]string	These are the body form params usually referred as PostParams.	

Functions Added by framework:

SNo	Function	Explanation	Example
1	Json(object)	The param can be anything. This function will convert the object to Json	
2	Object(id) Id – The id of the object to be loaded	This loads the object of the given id from the DB or from loaded object ➢ In case of flow identifier this will only use the loaded object and will not load from DB. So make sure you have added the object in flow objects ➢ For flow config and api responses this will try to load from loaded object and if not found it will load from DB	Object 1 Loads the object with Id 1
3	Get (JsonObject, key1, key2…. keyn)	This is a getter for this Json object. ➢ For dictionary or Objects the keys are the key ➢ For the Array the keys are the index values. Index starts from 1 to N	Get (Object 1) “account” “accountId”

	JsonObject – A json object type keyn – This can be either the key of the object or index incase of array	This function either returns ➢ Json Object (for array or Obj/Dict) ➢ Bool ➢ String ➢ Number (float64) ➢ Nil in case of nonexistent key or null value	This operates on the Object 1 and gets Object.account.accountId value
4	Set (JsonObject, key1, key2… keyn, value) JsonObject – A json object type keyn – This can be either the key(string) of the object or index(integer) incase of array value – the value to be set for key1.key2…keyn	Simillar to Get this sets the value for the object in key1.Key2….Keyn PS: this only updates the object in the memory. To persist it call the SaveObject For array the key can be positive or negative ➢ Index > 0 (positive), The value gets replaced in that index. If the index is more than the length of the array then it gets added to the end of the array ➢ Index is 0, it gets added to the start of the array ➢ Index < 0 (negative) , the value will be inserted in that index(after making it positive) position. If the absolute index is more than the length of the array then it inserts at the end of the array	Set (Object 1) "loan" "completedSteps" 1 "AR"
5	Del (JsonObject, key1, key2…. keyn) JsonObject – A json object type keyn – This can be either the key of the object or index incase of array	This deletes the last key ie., keyn ➢ In case the keyn-1 is a object then this removes the key keyn from it ➢ In case the keyn-1 is an array then it deletes the keyn(int) index from the array. Ps: Index starts from 1 to N. Also if the index < 1 or index>N then it does not do anything	Del (Object 1) "loan" "completedSteps" 1
6	SaveObject (id)	Save the object with any changes made to the DB	SaveObject 1 Save the object 1 to db
7	Copy (JsonObj) JsonObject – A json object type	This just creates a deep copied object of the JsonObj that was passed. Changes to the new object will not affect the passed one	Copy (Object 1)
8	NewJsonArr	Creates a new empty json array	
9	NewJsonObj	Creates a new empty json dict/object	
10	add(a,b,..) a,b – can be any type of number	Adds multiple numbers. a+b+…. Ps: will neglect non numbers	add 1 2 3.5 6 returns 12.5
11	subract(a,b,…)	a-b-c-….	subtract 5 3.5 2 4

	a,b- can be any type of number		returns -4.5
12	minus(a) a – any type of number	Return the additive inverse of a number (-1 * number) This can be useful while chaining with add instead of using subtract	minus 4 retruns -4 add 3 4 6 (minus 5) 2 returns 10
13	multiply(a,b,…) a,b, - can be a number of any type	a*b*…	multiply 5 3.5 2 return 35
14	divide(a,b,…) a,b, - can be a number of any type	((a/b)/c) ….	divide 100 2.5 4 returns 10

15	oneby(a)
	
	a – any type of number
		Return the multiplicative inverse of a number (1/number)
		
		This can be useful while chaining with multiply instead of using subtract
			oneby 5
			
			return 0.2
			
			multiply 100 (oneby 2) 3.5
			
			return 175

Extracts from text/template: (useful reading items)

```
{{if pipeline}} T1 {{end}}
        If the value of the pipeline is empty, no output is generated;
        otherwise, T1 is executed. The empty values are false, 0, any
        nil pointer or interface value, and any array, slice, map, or
        string of length zero.
        Dot is unaffected.

{{if pipeline}} T1 {{else}} T0 {{end}}
        If the value of the pipeline is empty, T0 is executed;
        otherwise, T1 is executed. Dot is unaffected.

{{if pipeline}} T1 {{else if pipeline}} T0 {{end}}
        To simplify the appearance of if-else chains, the else action
        of an if may include another if directly; the effect is exactly
        the same as writing
                {{if pipeline}} T1 {{else}}{{if pipeline}} T0 {{end}}{{end}}
```

Arguments

An argument is a simple value, denoted by one of the following.

```
- A boolean, string, character, integer, floating-point, imaginary
  or complex constant in Go syntax. These behave like Go's untyped
  constants. Note that, as in Go, whether a large integer constant
  overflows when assigned or passed to a function can depend on whether
  the host machine's ints are 32 or 64 bits.
- The keyword nil, representing an untyped Go nil.
- The character '.' (period):
        .
  The result is the value of dot.
- A variable name, which is a (possibly empty) alphanumeric string
  preceded by a dollar sign, such as
        $piOver2
  or
        $
  The result is the value of the variable.
  Variables are described below.
- The name of a field of the data, which must be a struct, preceded
  by a period, such as
        .Field
  The result is the value of the field. Field invocations may be
  chained:
    .Field1.Field2
  Fields can also be evaluated on variables, including chaining:
    $x.Field1.Field2
- The name of a key of the data, which must be a map, preceded
  by a period, such as
        .Key
  The result is the map element value indexed by the key.
  Key invocations may be chained and combined with fields to any
  depth:
    .Field1.Key1.Field2.Key2
  Although the key must be an alphanumeric identifier, unlike with
  field names they do not need to start with an upper case letter.
  Keys can also be evaluated on variables, including chaining:
    $x.key1.key2
- The name of a niladic method of the data, preceded by a period,
  such as
        .Method
  The result is the value of invoking the method with dot as the
  receiver, dot.Method(). Such a method must have one return value (of
  any type) or two return values, the second of which is an error.
  If it has two and the returned error is non-nil, execution terminates
  and an error is returned to the caller as the value of Execute.
  Method invocations may be chained and combined with fields and keys
  to any depth:
    .Field1.Key1.Method1.Field2.Key2.Method2
  Methods can also be evaluated on variables, including chaining:
    $x.Method1.Field
- The name of a niladic function, such as
        fun
  The result is the value of invoking the function, fun(). The return
  types and values behave as in methods. Functions and function
  names are described below.
- A parenthesized instance of one the above, for grouping. The result
  may be accessed by a field or map key invocation.
        print (.F1 arg1) (.F2 arg2)
        (.StructValuedMethod "arg").Field
```

Arguments may evaluate to any type; if they are pointers the implementation automatically indirects to the base type when required. If an evaluation yields a function value, such as a function-valued field of a struct, the function is not invoked automatically, but it can be used as a truth value for an if action and the like. To invoke it, use the call function, defined below.

Examples

Here are some example one-line templates demonstrating pipelines and variables. All produce the quoted word "output":

```
{{"\"output\""}}
        A string constant.
{{`"output"`}}
        A raw string constant.
{{printf "%q" "output"}}
        A function call.
{{"output" | printf "%q"}}
        A function call whose final argument comes from the previous
        command.
{{printf "%q" (print "out" "put")}}
        A parenthesized argument.
{{"put" | printf "%s%s" "out" | printf "%q"}}
        A more elaborate call.
```

```
{{"output" | printf "%s" | printf "%q"}}
        A longer chain.
{{with "output"}}{{printf "%q" .}}{{end}}
        A with action using dot.
{{with $x := "output" | printf "%q"}}{{$x}}{{end}}
        A with action that creates and uses a variable.
{{with $x := "output"}}{{printf "%q" $x}}{{end}}
        A with action that uses the variable in another action.
{{with $x := "output"}}{{$x | printf "%q"}}{{end}}
        The same, but pipelined.
```

Variables

A pipeline inside an action may initialize a variable to capture the result. The initialization has syntax

```
$variable := pipeline
```

where $variable is the name of the variable. An action that declares a variable produces no output.

Variables previously declared can also be assigned, using the syntax

```
$variable = pipeline
```

If a "range" action initializes a variable, the variable is set to the successive elements of the iteration. Also, a "range" may declare two variables, separated by a comma:

```
range $index, $element := pipeline
```

in which case $index and $element are set to the successive values of the array/slice index or map key and element, respectively. Note that if there is only one variable, it is assigned the element; this is opposite to the convention in Go range clauses.

A variable's scope extends to the "end" action of the control structure ("if", "with", or "range") in which it is declared, or to the end of the template if there is no such control structure. A template invocation does not inherit variables from the point of its invocation.

When execution begins, $ is set to the data argument passed to Execute, that is, to the starting value of dot.

Functions

During execution functions are found in two function maps: first in the template, then in the global function map. By default, no functions are defined in the template but the Funcs method can be used to add them.

Predefined global functions are named as follows.

```
and
        Returns the boolean AND of its arguments by returning the
        first empty argument or the last argument, that is,
        "and x y" behaves as "if x then y else x". All the
        arguments are evaluated.
call
        Returns the result of calling the first argument, which
        must be a function, with the remaining arguments as parameters.
        Thus "call .X.Y 1 2" is, in Go notation, dot.X.Y(1, 2) where
        Y is a func-valued field, map entry, or the like.
        The first argument must be the result of an evaluation
        that yields a value of function type (as distinct from
        a predefined function such as print). The function must
        return either one or two result values, the second of which
        is of type error. If the arguments don't match the function
        or the returned error value is non-nil, execution stops.
html
        Returns the escaped HTML equivalent of the textual
        representation of its arguments. This function is unavailable
        in html/template, with a few exceptions.
index
        Returns the result of indexing its first argument by the
        following arguments. Thus "index x 1 2 3" is, in Go syntax,
	x[1][2][3]. Each indexed item must be a map, slice, or array.
slice
	slice returns the result of slicing its first argument by the
	remaining arguments. Thus "slice x 1 2" is, in Go syntax, x[1:2],
	while "slice x" is x[:], "slice x 1" is x[1:], and "slice x 1 2 3"
	is x[1:2:3]. The first argument must be a string, slice, or array.
js
	Returns the escaped JavaScript equivalent of the textual
	representation of its arguments.
len
	Returns the integer length of its argument.
not
	Returns the boolean negation of its single argument.
or
	Returns the boolean OR of its arguments by returning the
	first non-empty argument or the last argument, that is,
	"or x y" behaves as "if x then x else y". All the
	arguments are evaluated.
print
	An alias for fmt.Sprint
printf
	An alias for fmt.Sprintf
println
	An alias for fmt.Sprintln
urlquery
	Returns the escaped value of the textual representation of
	its arguments in a form suitable for embedding in a URL query.
	This function is unavailable in html/template, with a few
	exceptions.
```

The boolean functions take any zero value to be false and a non-zero value to be true.

There is also a set of binary comparison operators defined as functions:

```
eq
	Returns the boolean truth of arg1 == arg2
ne
	Returns the boolean truth of arg1 != arg2
lt
	Returns the boolean truth of arg1 < arg2
le
	Returns the boolean truth of arg1 <= arg2
gt
	Returns the boolean truth of arg1 > arg2
ge
	Returns the boolean truth of arg1 >= arg2
```

For simpler multi-way equality tests, eq (only) accepts two or more arguments and compares the second and subsequent to the first, returning in effect

```
arg1==arg2 || arg1==arg3 || arg1==arg4 ...
```

Examples:

```
    {{$obj := Object 5}}  Assigning the result of the function to $obj

    {{if eq "NOT_LINKED" (Get $obj "account" "accountStatus")}} if and equal

    {{Set $obj "account" "accountStatus" "LINKED"}} All are arguments of function Set. Set is a
framework defined function

    {{Set $obj "loan" "eligibility" "pendingUserData" 1 "BASIC"}} Example of mixed arguments for
the set function



    {
        "responseStatus": "SUCCESS",
        "data":
        {
                "account" : {{ Json (Get $obj "account")}},
                "loan" : {{ Json (Get $obj "loan")}}
        }
    }
This portion is just a combination of normal text and template. The normal text (outside {{ }} )
will be evaluated/printed as is. Whereas those inside {{}} will be evaluated as a template



    {{else}} else of the if above. There can be {{else if <condition>}} too

    {{$invalid_stage := Copy (Object 9)}}  another variable

    {{Set $invalid_stage "error" "description" (print "Expecting user data for " (Get $obj "loan"
"eligibility" "pendingUserData" 1))}} see chaining of methods used as argument to another method.
print is the builtin function which can be used to concat string

    {{end}}
```

Appendix-1

Example flow in Yaml

```
config:
  GET /KISSHT/accounts/:accountId/account:
    status: ''
    header: ''
    response_body: '
    {{$obj := Object 5}}
    {{if eq "NOT_LINKED" (Get $obj "account" "accountStatus")}}
    {{$invalid_stage := Copy (Object 9)}}
    {{Set $invalid_stage "data" "account" "accountId" .PathParams.accountId}}
    {{Set $invalid_stage "data" "account" "accountStatus" "NOT_LINKED"}}
    {{Set $invalid_stage "data" "account" "reason" "INVALID_STAGE - Account not linked!"}}
    {{Set $invalid_stage "error" "description" "Account not linked!"}}
    {{Json $invalid_stage}}
    {{else}}
    {
        "responseStatus": "SUCCESS",
        "data":
        {
                "account" : {{ Json (Get $obj "account")}}
        }
    }
    {{end}}
      '
  GET /KISSHT/accounts/:accountId/loan:
    status: ''
    header: ''
    response_body:
    response_body: '
    {{$obj := Object 5}}
    {{if eq "NOT_LINKED" (Get $obj "account" "accountStatus")}}
    {{$invalid_stage := Copy (Object 9)}}
    {{Set $invalid_stage "data" "account" "accountId" .PathParams.accountId}}
    {{Set $invalid_stage "data" "account" "accountStatus" "NOT_LINKED"}}
    {{Set $invalid_stage "data" "account" "reason" "INVALID_STAGE - Account not linked!"}}
    {{Set $invalid_stage "error" "description" "Account not linked!"}}
    {{Json $invalid_stage}}
    {{else}}
    {
        "responseStatus": "SUCCESS",
        "data":
        {
                "account" : {{ Json (Get $obj "account")}},
                "loan" : {{ Json (Get $obj "loan")}}
        }
    }
    {{end}}
      '
  GET /KISSHT/accounts/:accountId/loanUser:
    status: ''
    header: ''
    response_body: '
    {{$obj := Object 5}}
    {{if eq "NOT_LINKED" (Get $obj "account" "accountStatus")}}
    {{$invalid_stage := Copy (Object 9)}}
    {{Set $invalid_stage "data" "account" "accountId" .PathParams.accountId}}
    {{Set $invalid_stage "data" "account" "accountStatus" "NOT_LINKED"}}
    {{Set $invalid_stage "data" "account" "reason" "INVALID_STAGE - Account not linked!"}}
    {{Set $invalid_stage "error" "description" "Account not linked!"}}
    {{Json $invalid_stage}}
    {{else}}
    {
        "responseStatus": "SUCCESS",
        "data":
        {
                "account" : {{ Json (Get $obj "account")}},
                "loan" : {{ Json (Get $obj "loan")}},
                "user" : {{ Json (Get $obj "user")}}
        }
    }
    {{end}}
```

```
      '
  POST /KISSHT/accounts/:accountId/link:
    status: ''
    header: ''
    response_body: '
    {{$obj := Object 5}}

    {{if eq "NOT_LINKED" (Get $obj "account" "accountStatus")}}
    {{Set $obj "account" "accountStatus" "LINKED"}}
    {{Set $obj "loan" "eligibility" "pendingUserData" 1 "BASIC"}}
    {{Set $obj "loan" "eligibility" "status" "PENDING"}}

    {
        "responseStatus": "SUCCESS",
        "data":
        {
                "account" : {{ Json (Get $obj "account")}},
                "loan" : {{ Json (Get $obj "loan")}}
        }
    }
    {{Set $obj "account" "accountStatus" "NOT_LINKED"}}

    {{SaveObject 5}}
    {{else}}
    {{$invalid_stage := Copy (Object 9)}}
    {{Set $invalid_stage "data" "account" "accountId" .PathParams.accountId}}
    {{Set $invalid_stage "data" "account" "accountStatus" "LINKED"}}
    {{Set $invalid_stage "data" "account" "reason" "INVALID_STAGE - Account already linked!"}}
    {{Set $invalid_stage "error" "description" "Account already linked!"}}
                {{Json $invalid_stage}}
    {{end}}
    '
  POST /KISSHT/accounts/:accountId/eligibility/:userDataSet:
    status: ''
    header: ''
    response_body: '
    {{$obj := Object 5}}
    {{$invalid_stage := Copy (Object 9)}}
    {{Set $invalid_stage "data" "account" "accountId" .PathParams.accountId}}


    {{if eq "NOT_LINKED" (Get $obj "account" "accountStatus")}}
    {{if eq .PathParams.userDataSet "BASIC"}}
    {{Set $obj "account" "accountStatus" "ELIGIBLE"}}
    {{Set $obj "loan" "eligibility" "pendingUserData" (NewJsonArr)}}
    {{Set $obj "loan" "eligibilityAmountX100" 10000000}}
    {{Set $obj "loan" "availableAmountX100" 10000000}}
    {{Set $obj "loan" "eligibility" "status" "SUCCESS"}}
    {{Set $obj "loan" "score" "820"}}
    {{Set $obj "loan" "tier" "GOLD"}}

    {
        "responseStatus": "SUCCESS",
        "data":
        {
                "account" : {{ Json (Get $obj "account")}},
                "loan" : {{ Json (Get $obj "loan")}}
        }
    }
    {{Set $obj "user" .JSON}}
    {{else}}
    {{Set $invalid_stage "data" "account" "accountStatus" "LINKED"}}
    {{Set $invalid_stage "data" "account" "reason" (print "INVALID_STAGE - Expecting user data
for " (Get $obj "loan" "eligibility" "pendingUserData" 1))}}
    {{Set $invalid_stage "error" "description" (print "Expecting user data for " (Get $obj "loan"
"eligibility" "pendingUserData" 1))}}
    {{Json $invalid_stage}}
    {{end}}
    {{else}}
    {{Set $invalid_stage "data" "account" "accountStatus" "ELIGIBLE"}}
    {{Set $invalid_stage "data" "account" "reason" "INVALID_STAGE - Account already eligible!"}}
    {{Set $invalid_stage "error" "description" "Account already eligible!"}}
    {{Json $invalid_stage}}
    {{end}}
    '
identifier: '{{if eq .PathParams.accountId (Get (Object 5) "account" "accountId")}}true{{end}}'
objects:
- 5
- 9
title: Link Basic Flow
```