

# How to Perform API Testing with Test Case Designer

This article explains how Test Case Designer can be used to improve API testing with 2 examples from CRUD operations. As a quick reminder of our applicability guidelines and standard benefits – if a system or process contains several decision points and at least 2 steps with multiple options per step, then TCD can be used to achieve the benefits of:

- improved defect prevention,
- time savings, and
- better coverage understanding.

Clients often experience these benefits first with their functional UI testing projects. It can take more time for teams to realize that those same benefits also apply to API testing. While all of the above benefits remain relevant, many API teams especially appreciate the improved testing thoroughness that Test Case Designer ensures.

As shown in the second use case below, when you leverage the tool to design your API scenarios, you won't stop with a simple shallow test that confirms that it is possible to "retrieve a member's profile." You'll always systematically generate far more robust, thorough tests that verify that retrieving a member profile will work correctly regardless of the member enrollment plan type, effective status, delivery type, address format, or other characteristics a member has.

- [POST](#)
- [GET](#)
- [Conclusion](#)

## POST

Let's start with the simpler testing objective – submitting a payment through [an API like this](#) that accepts card transactions for different types, currencies, amounts with/without discounts, etc.






The request structure for many POST operations clearly outlines parameter and value candidates:

```
function initPaymentRequest() {
  let networks = ['amex', 'diners', 'discover', 'jcb', 'mastercard', 'unionpay',
    'visa', 'mir'];
  let types = ['debit', 'credit', 'prepaid'];
  let supportedInstruments = [{
    supportedMethods: 'basic-card',
    data: {supportedNetworks: networks, supportedTypes: types},
  }];

  let details = {
    total: {label: 'Donation', amount: {currency: 'USD', value: '55.00'}},
    displayItems: [
      {
        label: 'Original donation amount',
        amount: {currency: 'USD', value: '65.00'},
      },
      {
        label: 'Friends and family discount',
        amount: {currency: 'USD', value: '-10.00'},
      },
    ],
  },
];
```

To limit the scope for example purposes, let's say we act as a vendor with 90% of payments coming from North America and the 4 major card networks. Further, the discount is driven by the amount alone being higher than 100 (regardless of the currency). With those assumptions, the initiation of your TCD model could look like this:



Card Network (5)	"amex"	"discover"	"mastercard"	"visa"	other 
Card Type (3)	Debit	Credit	Prepaid		
Currency (3)	"USD"	"CAD"	other 		
Amount (3)	with discount 	without discount - medium 	without discount - min 		
Discount (2)	"0"	"-10.00"			

Value expansions cover card types and currencies we rarely experience in production. They also add extra variety to the amounts for boundary and format testing. We can then add this rule – "Amount[with discount] <-> Discount["-10.00"]" – and see how TCD quickly generates the scenarios:



15 scenarios and 97 2-way interactions



#	Card Network	Card Type	Currency	Amount	Discount
1	"amex"	Debit	"USD"	"100.00" - wit... medium	"0"
2	"discover"	Credit	"CAD"	"100.50" - with discount	"-10.00"
3	"mastercard"	Prepaid	"EUR" - other	"10" - without discount - min	"0"
4	"visa"	Debit	"GBP" - other	"150" - with discount	"-10.00"
5	"jcb" - other	Prepaid	"USD"	"100.50" - with discount	"-10.00"
6	"visa"	Credit	"CAD"	"75" - without... medium	"0"
7	"unionpay" - other	Debit	"CAD"	"34.99" - with...t - min	"0"
8	"amex"	Credit	"JPY" - other	"10" - without discount - min	"0"
9	"discover"	Prepaid	"EUR" - other	"100.00" - wit... medium	"0"
10	"mastercard"	Credit	"USD"	"150" - with discount	"-10.00"
11	"amex"	Prepaid	"CAD"	"100.50" - with discount	"-10.00"
12	"discover"	Debit	"USD"	"34.99" - with...t - min	"0"
13	"mastercard"	Debit	"CAD"	"75" - without... medium	"0"
14	"visa"	Prepaid	"USD"	"10" - without discount - min	"0"
15	"diners" - other	Credit	"GBP" - other	"100.00" - wit... medium	"0"



Keep in mind that **only** value expansions are populated in the export (not “value – value expansion” syntax), so they won’t interfere with the scripting for execution – i.e. the csv table or the Automate script would have only “100.00”, not “100.00 – without discount – medium”.

This example can be expanded to support shipping and/or contact information alongside other payment aspects – add those parameters to the model.

## GET

Now let’s talk about the trickier objective – testing the API that retrieves the details of the insurance policyholder. The request body often contains just one field – Member ID. So, building a Test Case Designer model with that one parameter wouldn’t make sense, right? Yes and no.

While the request body lacks variety, we are more interested in what we are getting back – the ability to retrieve the member details correctly regardless of the member type, address format, other characteristics. So, we create the Test Case Designer model based on the important factors from the **response** structure, not the request one. The corresponding simplified scope could look like this:

API Demo - GET member details ▾



Save (Cmd+S)

```
1 Channel Sending the Request [ Internal Channel 1 , Internal Channel 2 , External Partner ]
2 Member Profile Details [ ***** ]
3 Relationship [ Primary policy holder , Spouse , Child ]
4 Member Enrollment Plan [ Plan Type 1 , Plan Type 2 ]
5 Term Effective Date [ past , current , future ]
6 Effective Status [ Active , Terminated within grace period , Terminated outside of grace period ]
7 Delivery Preferences [ true , false ]
8 Address [ domestic - single line , domestic - multi line , international ]
```

For some parameters, the values are not obvious and come from the format differences instead of the content ones. Typical examples include:

- addresses (single line vs. multi-line instead of “123 XYZ Street” vs. “345 ABC Drive”);
- dates (supporting both mm/dd/yyyy and dd/mm/yyyy formats, current or future terms instead of arbitrary “4/5/2021” vs. “5/4/2021”);
  - E.g., the “Term Effective Date” parameter above would have value expansions for formats.
- numbers (on top of any boundary conditions and rules, the validation of zero/N decimal spots should be considered).

Value expansions often play a “test data specifier” role in such models, as they contain actual text strings used in an execution.

However, such strategy may not be efficient for all value types from the maintenance perspective. Hardcoding “1/6/2022” as the “current” substitute will work for 1 day. For the execution to continue, it will need to be updated and re-exported regularly. Instead, we recommend keeping such values at the abstract level in TCD and having functions in the execution framework to replace them at runtime.

**For any CRUD operation type, you can also consider adding the “environmental” variables such as the channel the request is sent from, versioning of the protocol, etc.** You can see the draft-generated scenarios below:

15 scenarios and 169 2-way interactions									Freeze these Scenarios		Sa
#	Channel Sending the Request	Member Profile Details	Relationship	Member Enrollment Plan	Term Effective Date	Effective Status	Delivery Preferences	Address			
1	Internal Channel 1	*****	Primary policy holder	Plan Type 1	past in mm/dd/... - past	Active	true	domestic - single line			
2	Internal Channel 2	*****	Spouse	Plan Type 2	current in mm/...current	Active	false	domestic - multi line			
3	External Partner	*****	Child	Plan Type 1	past in dd/mm/... - past	Terminated within grace period	false	international			
4	External Partner	*****	Spouse	Plan Type 2	past in mm/dd/... - past	Terminated out... period	true	international			
5	Internal Channel 1	*****	Child	Plan Type 2	future in mm/d... future	Active	true	domestic - multi line			
6	Internal Channel 2	*****	Primary policy holder	Plan Type 1	past in dd/mm/... - past	Terminated out... period	false	domestic - single line			
7	External Partner	*****	Primary policy holder	Plan Type 2	past in mm/dd/... - past	Terminated within grace period	true	domestic - multi line			
8	Internal Channel 1	*****	Primary policy holder	Plan Type 1	current in dd/...current	Active	true	international			
9	External Partner	*****	Spouse	Plan Type 1	future in dd/m... future	Active	false	domestic - single line			
10	Internal Channel 2	*****	Child	Plan Type 2	past in dd/mm/... - past	Terminated within grace period	true	domestic - single line			
11	Internal Channel 1	*****	Spouse	Plan Type 1	past in mm/dd/... - past	Terminated out... period	false	domestic - multi line			
12	External Partner	*****	Child	Plan Type 1	current in mm/...current	Active	true	domestic - single line			
13	Internal Channel 2	*****	Primary policy holder	Plan Type 2	future in mm/d... future	Active	true	international			
14	Internal Channel 1	*****	Spouse	Plan Type 1	past in dd/mm/... - past	Terminated within grace period	true	domestic - multi line			
15	Internal Channel 1	*****	Child	Plan Type 1	past in mm/dd/... - past	Terminated out... period	true	domestic - multi line			

Typically, the **data dictionary section in the API documentation** provides a great start to building the Test Case Designer model. Just remember to consider both request and response variables, mandatory and optional, and think about the core underlying variation for each.

The remaining question – is how the script works when we can't put all these details into the request body. The following sequence will need to be followed:

- Find the record in the target system that meets the parameterized criteria for a given scenario;
- Collect the record ID;
- Pass it in the GET request;
- Validate the response against the parameterized criteria.

If the record does not exist, POST and GET could be tested in tandem from the same TCD model.



In some cases, the variety of the data you retrieve is limited by the vendor or other conditions. Therefore, identifying individual parameters and letting the Test Case Designer algorithm run “freely” lead to numerous combinations that cannot be created/obtained.

While you could try to mimic the test data restrictions with TCD constraints, the more efficient method would be to perform the import on the Forced Interactions screen combined with 1-way mixed-strength. And you would still get the benefit of one-to-many scripting during export.

## Conclusion

Hopefully, this article clarified a few perceived challenges regarding TCD applicability to non-UI types of testing.

“PUT” and “PATCH” methods can be modeled similarly to “POST”. Still, you should consider two groups of TCD parameters – group 1 reflecting the initial state and group 2 reflecting what group 1 is being changed to. “DELETE” is similar to “GET.”

When it comes to the transition from Test Case Designer to an automated execution framework like Ranorex, there are a couple of export options:

- Only the data table in JSON/CSV format;
- Gherkin scripts;
- Custom scripts tailored to your tool pipeline.