TestExpert

TestExpert version: 1.2.2
Document version: 1.2.2.1 (May 20, 2021)

This guide enables users to launch and become familiar with the TestExpert app and its features. Since the app is a Windows Store app this guide does not focus on installation: simply install it using Windows Store.

Topics

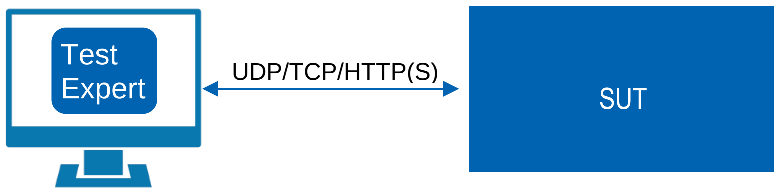
Introduction

Software organizations typically spend somewhere between 30 and 50% of the over-all development time on testing. These are tremendous figures and its therefore hard to understand why the problem of 'software testing' is still not getting the attention from software practitioners which it deserves. Real testing is the most important method to detect defects in the software design and is by far the only method which can identify defects that occur when a system is executing.

TestExpert is a tool which gives an answer on some of the questions which come up when dealing with software test:

- My application lives in a distributed environment and must implement a communication interface towards a server/client. How can I test my software against an external device which is still in development, doesn't support all of my features, or is too expensive to buy?
- How can I repeat previous tests when my software is getting into shape as a result of incremental development steps?
- How much can I adapt test cases for different test scenarios?

The tool allows a software developer or tester to define all kinds of test scenario's, execute them, and evaluate the test results. It interfaces with the *System Under Test* (SUT) through an IP network through which it exchanges message data either directly on UDP or TCP, or through HTTP.



With any of the supported protocols the user can define any type of test scenario: one where TestExpert will act as a client, one where it plays the server role, one where it plays both roles.

You can use TestExpert in the following 2 cases:

Communication Protocol Simulation/Emulation

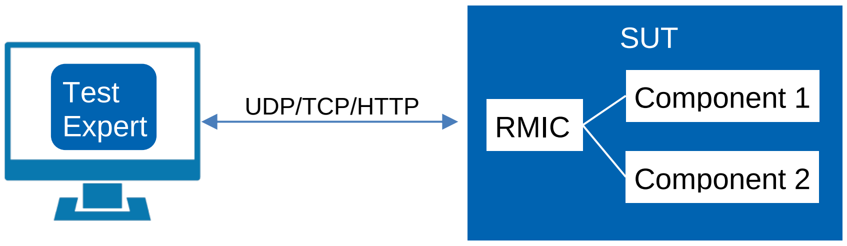
Without any extra components TestExpert can be directly used for communication protocol testing and simulation. To do this the app allows you to define any protocol message you want to run over UDP, TCP, or HTTP(S) using POST requests. Message elements can be specified in both binary and text based format and can be parameterized.

The definition of binary protocol messages is done through a message editor that allows you to define information element data as: byte, word, integer, long, string. You can also use expressions to create more complex data definitions.

Text based protocol messages can also easily be configured with the help of a built-in message editor. This editor derives the message elements by parsing template files containing a formal message syntax definition in ABNF notation.

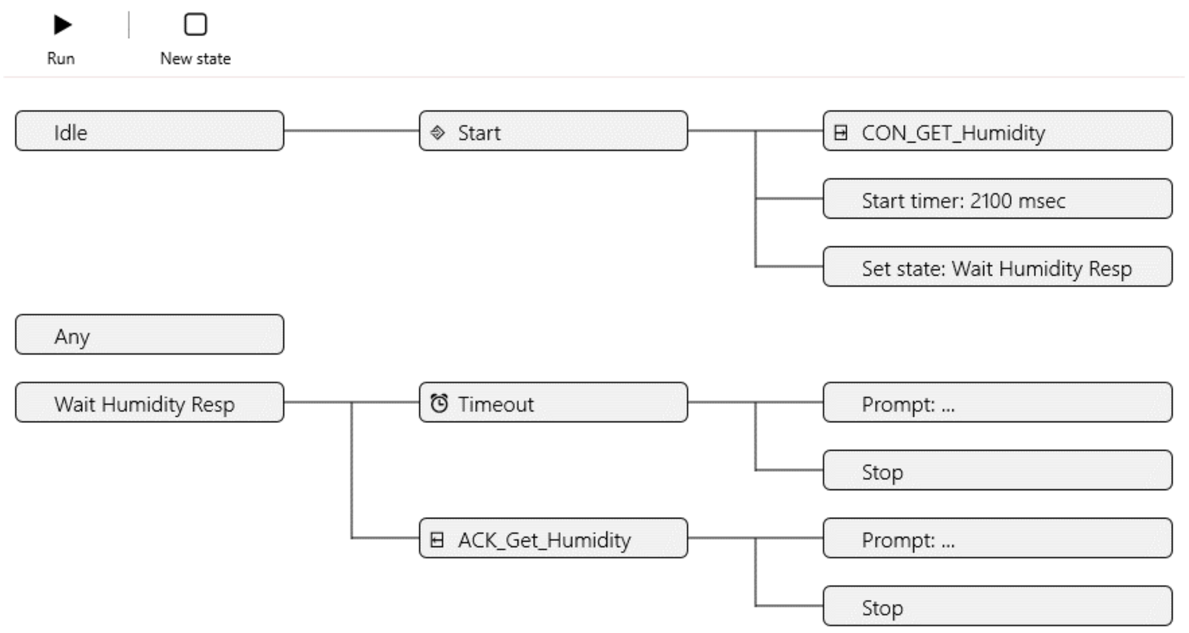
Software component test and integration test

With an additional *Remote Method Invocation Connector* (RMIC) TestExpert can also be used during software component test and integration test. The app then connects to the RMIC component on the SUT. The messages that will be passed over the UDP, TCP or HTTP(S) layer will then contain *Remote Procedure Call* data which will be exchanged with the target components/services through the RMIC.



Test Scenarios

TestExpert presents itself as a GUI application in which the user enters the test scenarios in the form of *State/Event/Action* notations.



- ‘States’ represent the different conditions in which a test is.
- ‘Events’ represent the signals which are being returned by the application or by TestExpert itself in a given state during the execution of the test.
- ‘Actions’ define the signals and the triggers which have to be delivered to the application or to TestExpert itself when a given event occurs.

During the execution of the test all reactions which are coming from the system, are being validated against what the tester is expecting in the momentary state of the scenario. When the reaction is correct then the scenario continues by triggering new actions in the system.

Execution of the test can both be done manually (one test at a time) or in batch mode (multiple scenarios are running after each other).

The possibility of TestExpert to evaluate the reaction of the system during the test and act accordingly, is considered to be a very important advantage when comparing with more traditional testing tools. These tools typically only generate a fixed set of signals without making any appreciation about what the system is supposed to be doing with it.

An Open Environment

In order to allow the use TestExpert on all kinds of applications, the tool has intentionally been designed around an ‘open’ concept. This means that the tool itself has no built-in knowledge of the application under test. It is the user’s task to define the interface protocols of the system that is to be tested.

By allowing the user to save the protocol elements and the test scenarios in so-called ‘solution’ files, all he protocol elements of a given application can be shared between different developers and testers.

TestExpert is a UWP app and therefore only runs on Windows 10 machines. The system under test of course can be any type of device as long as it supports UDP or TCP as communication interface with TestExpert and HTTP (in component test scenarios when RPC requests have to be exchanged through HTTP); i.e. Windows, iOS, Android, OS X, Linux, IOT devices, embedded systems, etc.

What you also should know

TestExpert presents itself as a low cost solution for doing what has been described above. That however doesn’t means that it can be a complete substitute for other, more expensive tools that also position themselves in the software testing area.

With TestExpert you don’t have a lot of out-of-the-box components to immediately let you start doing your tests. That means that there will be some effort involved from the user to define the various scenarios and all the messages that need to be exchanged with the external system.

TestExpert really works well for a programmer who is looking for a simple tool with which his or her software can connect at the moment a component, a system or a subsystem is ready for test. It is easy to setup, easy to modify, and intuitive.

The focus of the tool is in the low-level details of the actual testing process itself: the user-defined events and actions in test scenarios, the execution of the scenarios, and the possibility to adapt scenarios and messages to carry out alternate test cases. This also means that there is only basic logging, that there is no concept of traceability or test plans, and no management features.

Requirements

In order to use TestExpert in your test environment you need to take care of the following:

- TestExpert, being a UWP app, requires a Windows 10 machine. It is installed through the Windows Store.
- The system under test can be any device as long as it has the possibility to connect with the TestExpert app using UDP, TCP, or HTTP(S). The IP addresses, or the URL in case of HTTP(S), of the systems under test and the target/listening port numbers are fully configurable within TestExpert.

Component test

If you want to use TestExpert for component test then you need to develop a *Remote Method Invocation Connector* component and integrate/link it with your system. This component deals with the communication channel with TestExpert (using HTTP POST requests and HTTP responses), the de-serialization/serialization of method requests and responses, and the invocation of your component methods. Depending on whether your programming environment supports reflection or not this can be easily done or require some component-specific effort.

On the TestExpert website you can find a small library that implements a *Remote Method Invocation Connector* you can use when you are developing a Windows 10 UWP app. Connectors for other environments will be added when they become available from people willing to share their implementation.

Protocol Test

If you want to use TestExpert to emulate the communication protocol of a remote device then nothing else is needed. As already described UDP, TCP and HTTP are currently supported.

Network security protocols

Security at the IP layer (e.g. IPSec) or the upper layer (e.g. TLS) for UDP and TCP is considered for a possible later version.

HTTPS is supported in scenario’s where TestExpert is acting as a HTTP client; i.e. is sending HTTP request messages and expecting response messages.

When acting as a server only HTTP is supported.

Features

You can get detailed information about using TestExpert by selecting topics from the following list. Novice users should start with the items in the 'Basic principles' list: start with the first item, then the next, and so on. If you know the general concepts then you can get more detailed information about specific items by selecting a topic from the 'Advanced usage' list.

Basic principles

Topic	Description
The concept of scenarios	An explanation about what a scenario is and how it is central in the test concept.
Solution file	TestExpert stores scenarios and messages in a single solution file. The file can be used on any device that is running TestExpert.
Managing scenarios	A description about how you can create a scenario.
Managing messages	An explanation about the type of messages supported in TestExpert and how to add them.
Running a scenario	A description how you can execute a scenario

Advanced usage

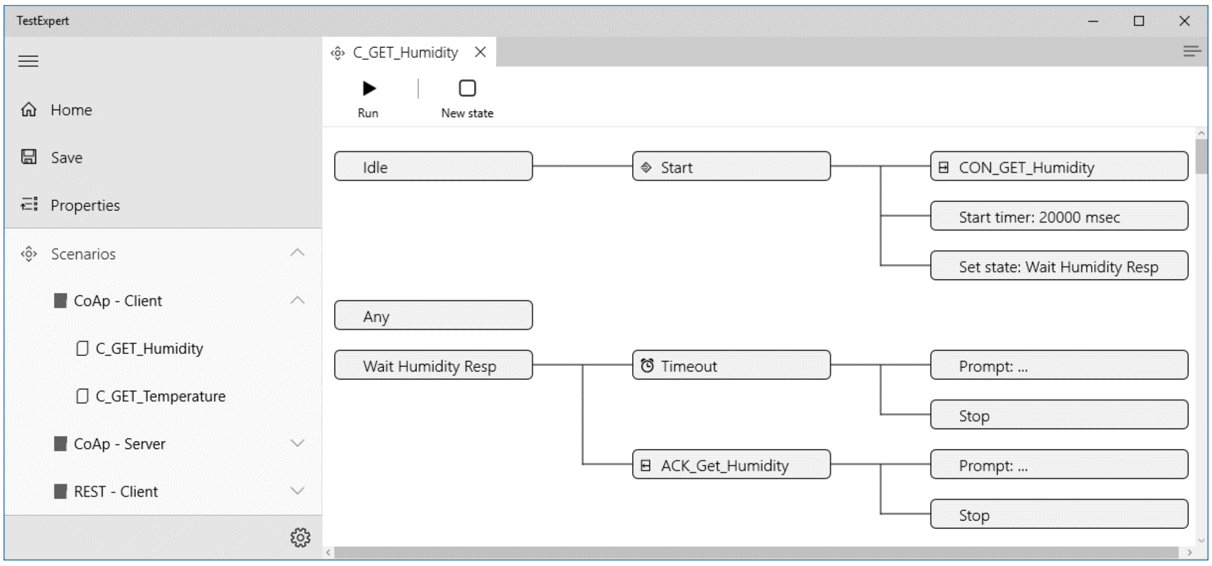
Topic	Description
Scenario editor	The Scenario editor lets you add, delete, and modify states, events, and actions in a scenario.
Message editor	The message editor lets you define the content of any type of message; binary or text based.
Message properties editor	With the *Message properties editor* you can configure some message properties and can verify whether the content is correctly specified.
Managing message elements	Message information elements are an additional feature of TestExpert to make it possible to include re-usable information in messages.
TestExpert variables	You cannot only define fixed content data in messages but also use data from Static variables and Runtime variables.
Solution properties page	This page lets you define some global solution settings like: endpoints, message encoding, etc.

The Concept of Scenarios

A characteristic of all kinds of testing, whether it is software component test, software integration test, or just testing of interfaces and protocols outside of a system, is the sequential nature of the different steps, events, and actions that have to occur during the test. TestExpert deals with this by presenting a concept to the user where he or she must create test scenarios in the form of *Finite State Machines (FSM)*. A scenario can be:

- Send a message to the target.
- Wait for a response message coming back.
- Validate the response.
- If the response is accepted, then send another message and wait for the next event.
- If the response is invalid, start a timer and stop the test after the timer expires.

Test scenarios are edited using TestExpert's Scenario Editor. The editor lets the user develop, modify and look at the scenario in a user friendly way.



A scenario contains: **states**, **events**, and **actions**.

- A **state** represents a halting point during execution of a test where TestExpert waits for an event to occur.
- An **event** is an internal or external trigger which happens at a certain moment within the momentary state.
  Events can be: an incoming message, a timeout, etc.. TestExpert allows you to specify the characteristics of these events: e.g. the incoming message contents, the timeout value, etc.
  With regard to incoming messages, TestExpert also allows you to extract values from the message and save them in so called 'run-time variables'. You can then import these variables within outgoing message definitions or use them to define conditional actions.
- An **action** constitutes a process of doing something.
  This can be: sending of a message, starting of a timer, changing the state, etc.

Also here, TestExpert will allow you to define the message contents, the timeout value, etc.
Actions also come with a *Condition* property with an expression that is validates in order to see if the action must execute or not.

Scenarios are presented (and specified) graphically on the screen in a kind of horizontal tree display with the states on the left, the events branching from the states and the actions branching from the events.

List of supported events

Event	Description
Start	An internally triggered event that occurs when execution of the scenario is initiated.
Timeout	An internally triggered event that occurs when a timer that is started as a result of a *Start timer* action has expired.
Incoming message	An event that occurs when a message is received on the communication channel. The event is validated by checking if the assigned message matches the received data. The assigned message (name, content) is configured by the user with the help of either a Basic message editor or a Template message editor.

List of supported actions

Action	Description
Start server	This action starts a communication listener service on a configured endpoint (network interface, listening port, etc.).
Send message	This action sends a user defined message out over the communication channel. The target address can be specified to be the default connected address or a specific one. The message itself (name, content) is configured by the user with the help of either a Basic message editor or a Template message editor.
Start timer	The *Start timer* action lets you tell the system to start a configurable timer.
Stop timer	This action stops any timer that is running.
Set state	The *Set state* action changes the current running scenario/FSM state to another state.

Give a prompt	*Give a prompt* allows you to bring a message up. Execution of the scenario is paused until the prompt is dismissed.
Call scenario	This action gives you the possibility to pause the currently running scenario, then execute another scenario, and later on (when the other scenario has finished execution) proceed with the main scenario.
Assign variable	*Assign variable* is an action that allows you to assign a value to a user-defined run-time variable. The variable can be used in a message definition.
Stop	This action stops execution of the scenario.

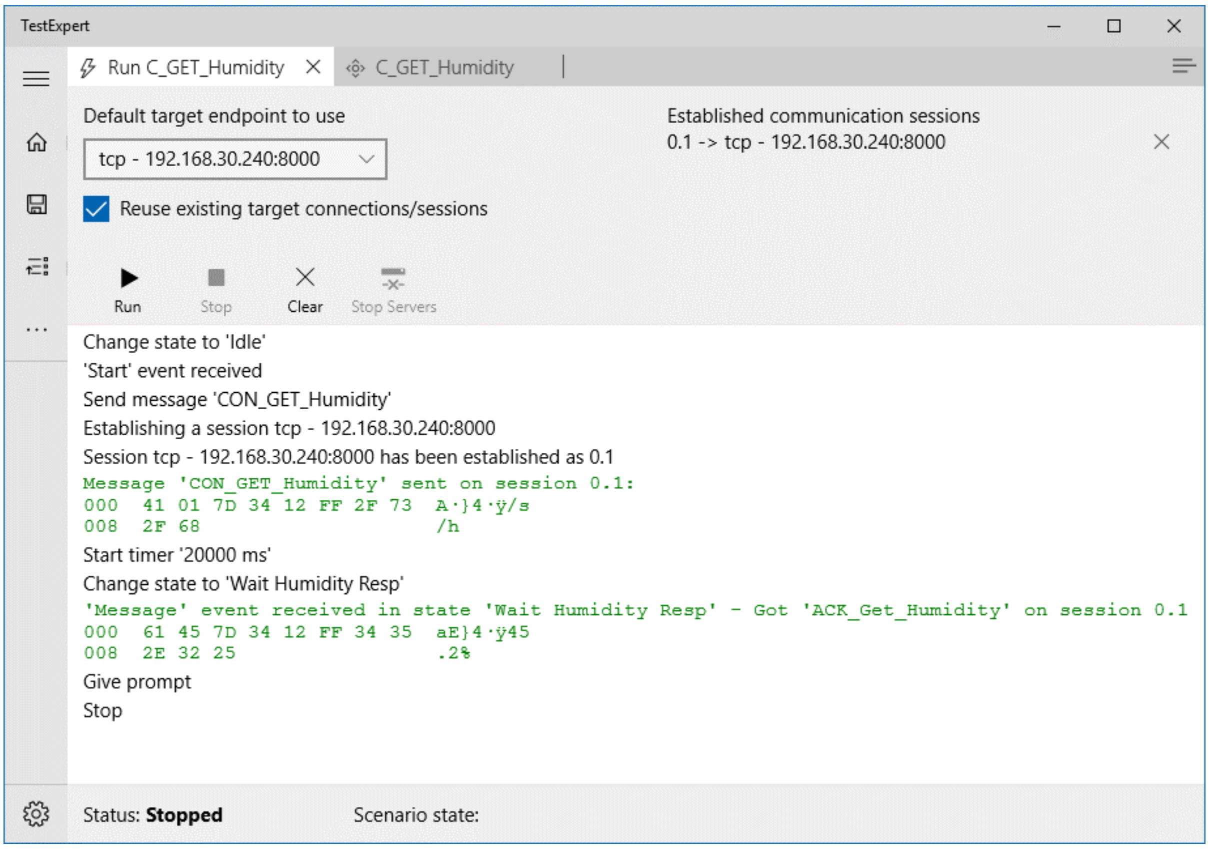
Execution of a scenario

When the user requests to start a scenario TestExpert will open a Scenario Execution View. Within that view the user can then request to run the scenario.

When execution is requested the state is set to *Idle* and a *Start* event is internally triggered.

- In a *Client* scenario the FSM will have actions assigned to the *Start* event, like: *Send message* (to send a message to the target), *Start timer* (to control receipt of a response, and *Set state* (to process the response message). TestExpert will handle those actions one after the other.
  When all actions are executed and the scenario is not stopped the *Scenario Handler* waits for a next event to occur.
  The process then repeats as soon as a new event comes in that matches the scenario state.
- In a *Server* scenario the FSM must also implement the *Start* event and attach a *Start server* action to it. This will instruct TestExpert to start a Server/Listener service on the configured listener endpoint.
  Because the target system in this setup will have to take the initiative to run all other scenario actions, the next thing you will have to do is to attach an *Incoming message* event to the scenario. You can do that on the *Idle* state and trigger execution of the scenario from there or you might introduce a new state that is set after the server is started and attach the message event to this new state.
  When the target system then sends the message, TestExpert checks whether it matches the *Incoming message* event and proceeds with executing the attached actions.

During execution of the scenario the view will be populated with information about everything that happens. Valid messages are shown in green, invalid and unrecognized messages are shown in red.



Default target endpoint to use: tcp - 192.168.30.240:8000

Reuse existing target connections/sessions ✓

Established communication sessions
0.1 -> tcp - 192.168.30.240:8000

```
Change state to 'Idle'
'Start' event received
Send message 'CON_GET_Humidity'
Establishing a session tcp - 192.168.30.240:8000
Session tcp - 192.168.30.240:8000 has been established as 0.1
Message 'CON_GET_Humidity' sent on session 0.1:
000  41 01 7D 34 12 FF 2F 73  A·}4·ÿ/s
008  2F 68                    /h
Start timer '20000 ms'
Change state to 'Wait Humidity Resp'
'Message' event received in state 'Wait Humidity Resp' - Got 'ACK_Get_Humidity' on session 0.1
000  61 45 7D 34 12 FF 34 35  aE}4·ÿ45
008  2E 32 25                 .2%
Give prompt
Stop
```

Status: **Stopped**   Scenario state:

Reuse of communication connections

When you run a scenario you have the possibility to re-use a possibly previously established communication channel to a target endpoint or create a new one. In order to allow such re-use TestExpert therefore will never automatically disconnect connections after a scenario has stopped.

The same is true with incoming server connections. TestExpert will keep all listening services that have been started active including all remote client connections.

The remote devices of course are free at any time to close their connection when they choose so. TestExpert will correctly handle that.

Solution File

A TestExpert solution maintains scenarios, messages, static variables, and various properties in a single *.txprt* solution file.
The *.txprt* file holds all the text-based information in XML format. This implies that, depending on how many scenarios and messages are defined for the solution, the file can grow from relatively small to quite big.

ABNF files that are used as template files for defining text based messages are separate files. When a solution uses them then the solution file only contains a reference to the actual physical ABNF file on a computer.

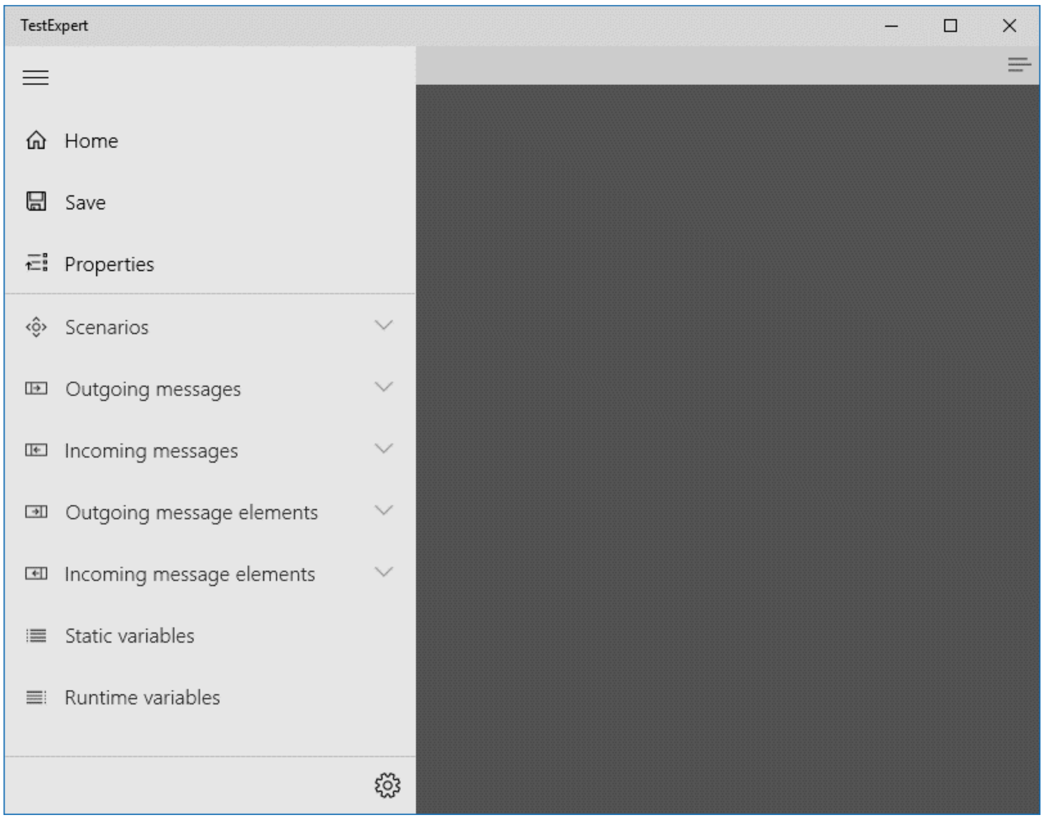
Create a new solution

You can create a new, blank solution when you are on the **Home** page of TestExpert. You get there automatically when TestExpert is started or by tapping on the *Home* button.

On the Home page, tap the *New solution* menu button.

A dialog window pops up where you must assign a name for the solution file and the folder where the file must be saved.

The solution view is then presented with a navigation menu showing on the left side.



The navigation menu lets you choose from the following items:

Icon	Menu	Description
	Home	Tap this menu item to go to TestExpert's **Home** page
	Save	Tap this menu item when you want to force TestExpert to save the solution file.
	Properties	The *Properties* menu item opens the Solution properties page.
	Scenarios	The *Scenarios* menu item lets you create, modify, rename or delete scenarios. You can group scenarios into folders and subfolders. See the section Managing scenario's for more detailed information.
	Outgoing messages *Incoming messages*	With these menu items you can create, modify, rename or delete outgoing resp. incoming messages. Information about defining messages can be found in the Managing messages chapter.
	Outgoing message elements *Incoming message elements*	These 2 menu items give you the possibility to define outgoing resp. incoming information elements for message of which the content is configured using the Raw message editor . These are elements which contain data values that you define once but include in multiple messages. See section Managing message elements for a full description.
	Static variables	Tap this item to open the Static Variables Editor. It allows you to create, modify and delete variables and assign values to them.
	Runtime variables	Tap this item to open the Runtime Variables Editor. You can then have a look at the variables that have been created dynamically while running a scenario.
	Settings	Tap this item to open the Runtime Variables Editor. You can then have a look at the variables that have been created dynamically while running a scenario.

If TestExpert is running in a small window then the navigation panel will be in a collapsed state. You can toggle between a full and a collapsed panel by tapping the top *Menu* button.

Open an existing solution

You can open an existing solution when you are on the **Home** page of TestExpert. You get there automatically when TestExpert is started or by tapping on the *Home* button.

You then have the option to select a solution from the recently opened files list or browse for a file by tapping the *Open file* action item.

Save solution

To save an open solution simply tap or click the *Save* menu button in the navigation panel.

Change solution properties

In addition to scenarios, messages and static variables a solution file also contains some global properties. You can configure these by tapping or clicking the Properties menu button.

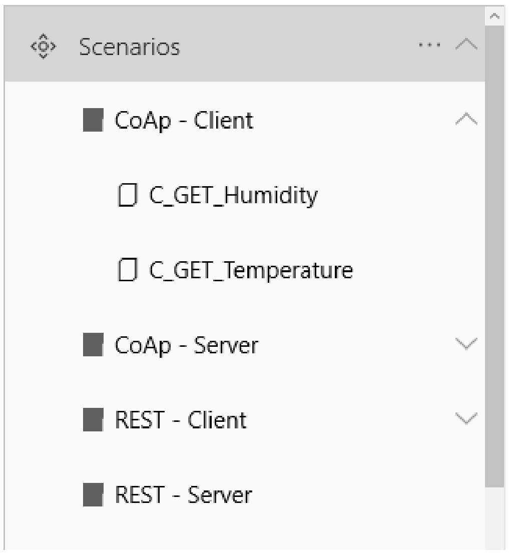
This opens the Solution properties page.

Open solution items

You must use the navigation pane to find folders and items within any of the main solution menu items.

To open the *Static variables* and *Runtime variables* items simply tap the corresponding menu item.

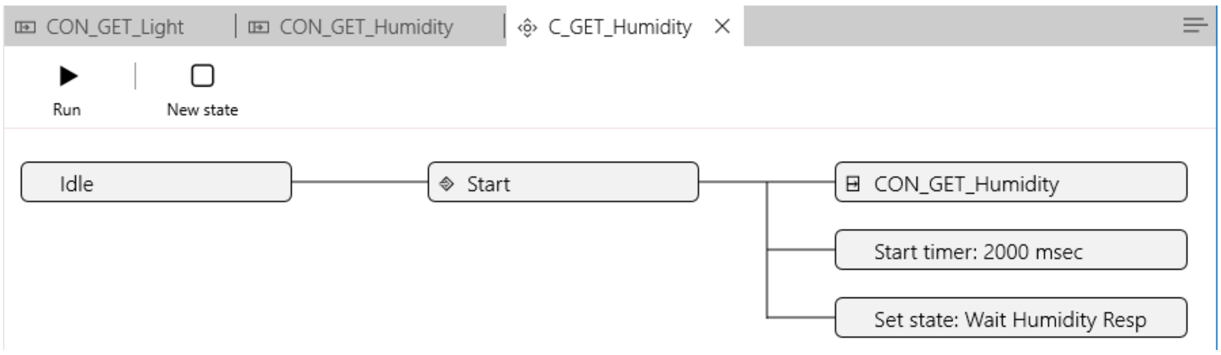
To open all other solution items (like those belonging to *Scenarios*, *Outgoing messages*, ..) you must first tap on the menu item to reveal the first level of folders and items. When an item is part of a subfolder then you have to proceed by tapping its parent folder until the item is revealed.



Tap then on the solution item to open the associated editor (a *Scenario editor*, M*essage editor*, etc.) in TestExpert's 'documents' pane. This pane hosts all open 'documents' either as individual tabs in the 'main' tab area (to the left) or as a single quick navigation tab in the 'preview' tab area (to the right).

Tabs area

Open solution items each have their own tab item in the tabs area. Only 1 item is shown at a time.



Note that a single tap in the navigation pane brings the item's editor in the 'preview' tab area. When you select another item afterwards then the previously shown item disappears and the new item takes its place. The 'preview' feature is there to allow for quick navigation through different solution items.

If you want an item to be in a persistent tab in the 'main' tab area then you can either double tap the item in the navigation pane or click somewhere in its editor.

There is no limit on how many solution items you can add to the 'main' tabs area. However, when there is insufficient space in the tab item's header bar, TestExpert will only show the most recently used tabs. The others are still there but hidden.

To reveal an item you have various options:

- Tap on the item's tab header

- Select the item in the navigation panel.
- Select the item in the tabs list. You can open the tabs list by tapping the list button to the right of the tab items bar.

  ☰

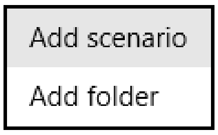
  You might want to use this method to show an item of which the tab item’s header is hidden due to space restrictions.

To close an item simply tap on the close button in the tab header itself. On an inactive tab the close button comes up when you place the mouse on its header.

Managing scenario’s

To create a new scenario tap the *More options* icon ··· to the right of the *Scenarios* menu item in the navigation panel.
Alternatively you can also right-click somewhere on the menu item bar.

Select *Add scenario* on the flyout menu.



TestExpert then adds a new scenario to the solution with a default name (change the name into something meaningful) and 2 pre-defined states: *Idle* and *Any*.

On the right side the Scenario editor is showing the new scenario/FSM.

The 2 pre-defined states are mandatory and can’t be deleted:

State	**Description**
Idle	This state represents the initial state of any scenario that is not yet running.
Any	This state can be used to assign events that are expected to occur in any state and for which you want to execute common actions when these events happen.

Use the Scenario editor now to setup your scenario with states, events and actions.

Folders and Subfolders

It is also possible to create a new scenario in a folder (or subfolder). To do that you must first create the folder/subfolder by selecting *Add folder* on the main *Scenarios* menu item or on an already existing folder item.
Then select the folder, bring up the flyout menu (using the *More options* icon or a right click) and tap *Add scenario* on the flyout menu.

Scenario editor

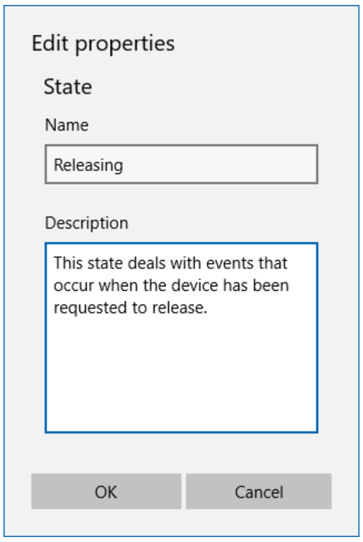
TestExpert's scenario editor allows you to setup and change a simulation scenario. You automatically invoke the editor each time you create a new scenario (see Create a scenario) or by opening an already existing scenario (see Open solution items).

Scenarios are edited interactively by the user by employing a set of buttons that are shown on the top tool bar, or selecting appropriate menu items from TestExpert's menu bar.. All buttons are logically arranged in groups.
The whole editing process keeps track of which item the user is editing by only enabling those tool bar buttons (showing them in full color) which are allowed to be pressed given a particular selected scenario item.

Adding States

A new State is added by pressing the *New state* button on the top toolbar.

TestExpert will pop-up a dialog window in which you can enter the name of the new state and a short description.



Name: Releasing

Description: This state deals with events that occur when the device has been requested to release.

State name

When entering a new state, you can choose whatever name you want except *Idle* and *Any*.

Examples

```
Releasing
Waiting_For_REL
Waiting for any message
```

Note that TestExpert has already pre-defined 2 states for you: state *Idle* and state *Any*.

- You must have the *Idle* state in your scenario because TestExpert assumes that this is the starting state for any test you want to conduct.
- As an option, you can also make use of the *Any* state in your scenario. This state has a special meaning for TestExpert during testing. When the test is running, TestExpert will always try to match an incoming event against the *Any* state when the normal match against the momentary state fails. This allows you to associate a set of events which you expect to happen in every state to the *Any* state.

Description

A description is optional. It is not used by TestExpert itself.

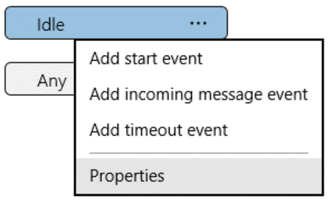
Adding additional states

You can of course create multiple states in the scenario by repeating the above process. Any new state is automatically added after the last created state.

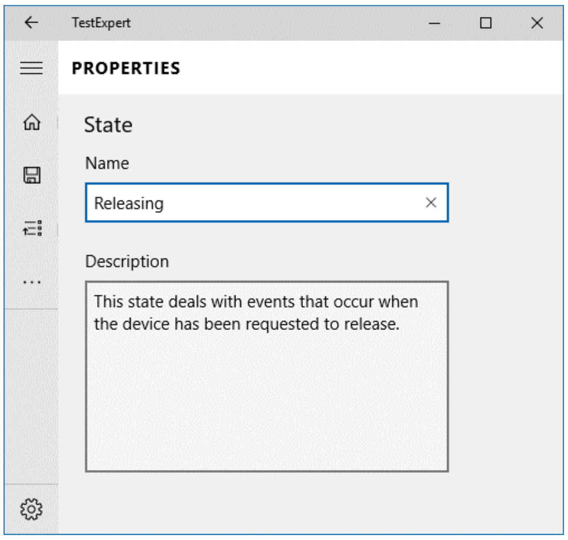
Although the order in which the state definitions appear is unimportant for TestExpert, you can move a state before or after another one by selecting the state (by clicking on it) and dragging the state rectangle to the wanted position.

State properties editor

You can at any time modify the selected state's name and description by means of the *State properties editor* that is also presented on the right side in case the view window is wide enough. If that is not the case then either tap the *More* menu button or right click on the state and select *Properties* in the flyout menu.



The *State properties editor* then pops up as a separate view page.



State

Name

Releasing

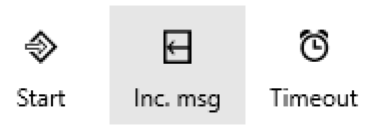
Description

This state deals with events that occur when the device has been requested to release.

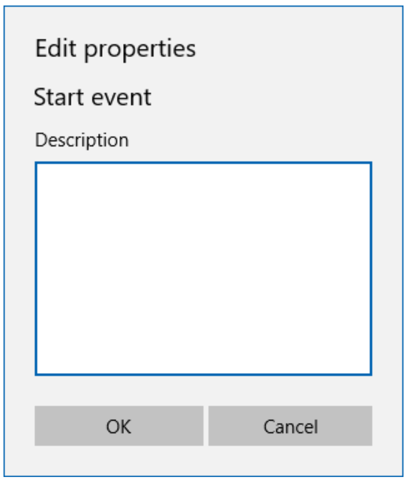
You can close the properties editor and go back to where you were by pressing the top left *Back* button.

Adding Events

The TestExpert scenario concept defines events as being internal and external triggers that happen in a given state. Events therefore are FSM elements that are attached to states.
You add them by first selecting the state (by tapping somewhere on the state symbol).
Once the state has been selected you can attach a specific event either by tapping on one of the event buttons in the top toolbar or tapping the menu button on the state and then selecting the event in the flyout menu.



You have the choice to attach 3 type of events to the state:

Event	Description	Event properties
Start	This event denotes an internally generated TestExpert trigger to start execution of the scenario. When you select this event a dialog pops up in which you can enter a description. Description Optional.	 Edit properties Start event Description OK Cancel

Incoming message

The purpose of the Incoming message event is to define an incoming message which TestExpert can expect from the target. When the scenario is running and has entered the given state, TestExpert will validate each incoming message in that state against the contents of the message which you have defined here.
The Incoming Message event can be attached to any state in the scenario. When attached to the Idle state, it typically indicates that TestExpert waits for an incoming message, sent by the target under test, to start the scenario activities.

When attaching this event you will be given the possibility to select the message, assign an optional source address and provide a description.

Message assignment

To select your message tap the dropdown box. It allows you to pick any of the already registered messages but also allows you to create the message at this point in time if that hasn't been done yet.

Note: If you create the message here you still have to specify the message content later on by bringing up the Message Editor.

Source address

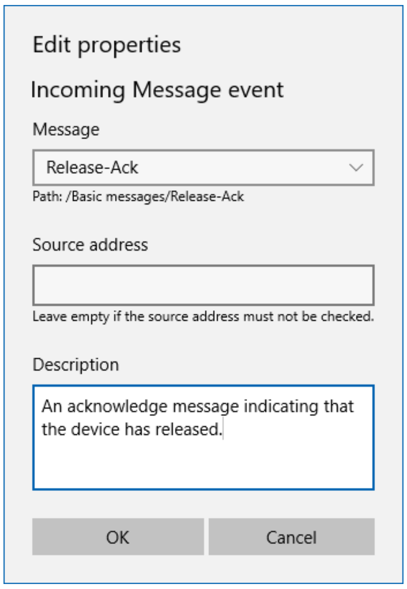
When the source address is left empty then TestExpert will treat any incoming message that occurs in the state as being a candidate for a match.

When the source address is defined then TestExpert will only match incoming messages that have a source address equal to the specified one.

The format of the source address is a network host address; e.g. An IP address.

Description

Optional.



Edit properties

Incoming Message event

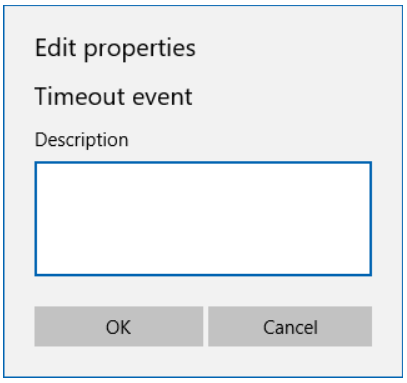
Message: Release-Ack

Path: /Basic messages/Release-Ack

Source address:

Leave empty if the source address must not be checked.

Description: An acknowledge message indicating that the device has released.

Timeout	This event indicates a possible timeout that can occur in the state to which the event is attached. Description Optional.	 Edit properties Timeout event Description

You can assign multiple events to the state by repeating the above process. Any new event is automatically added after the last event.

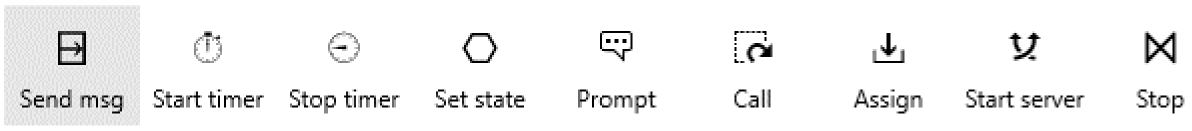
If you want to move an event before or after another one, then you can do this by selecting the event (by tapping on it) and dragging the mouse to the wanted position.

Event properties editor

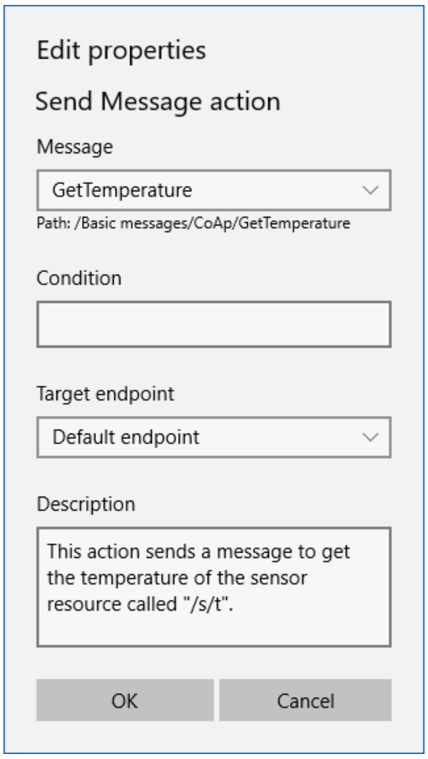
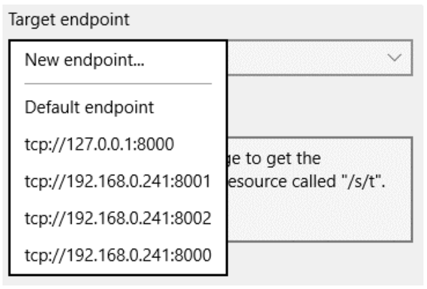
As with states you can also here modify the event properties (name, description, etc.) at any time by means of an *Event properties editor*. The editor is automatically displayed on the right side of the window when an event is selected and the view window is wide enough. If the windows is too narrow then tap the menu button on the event and select *Properties* in the flyout menu.

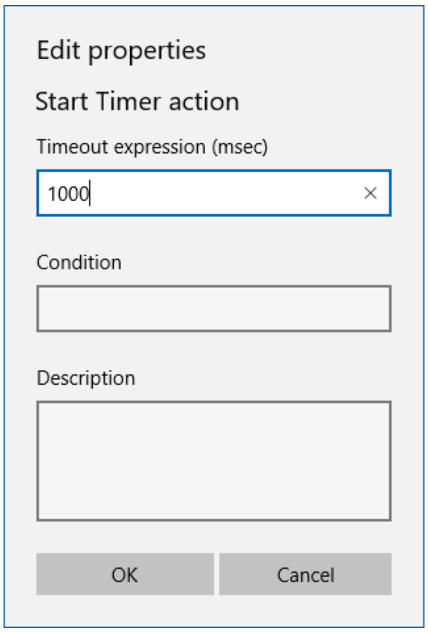
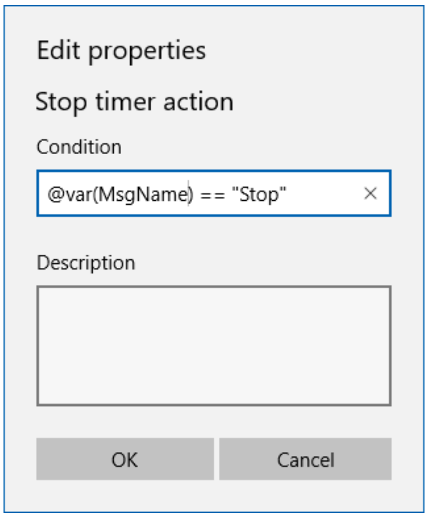
Adding Actions

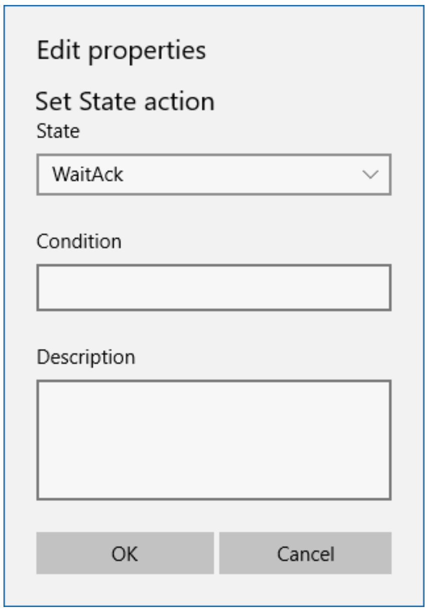
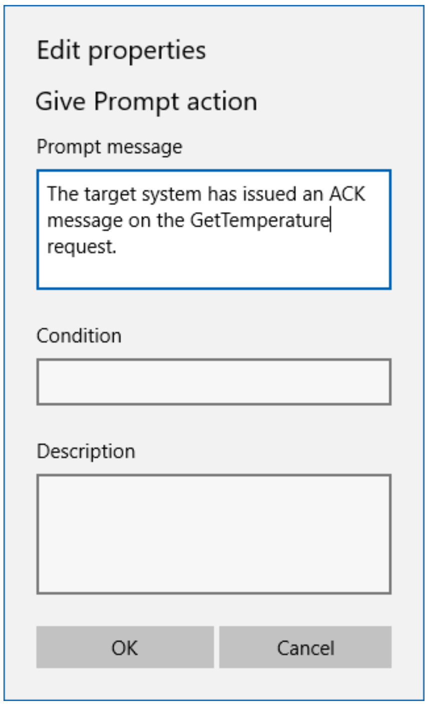
Actions define what TestExpert must do when the scenario runs and an event comes in which matches one of the events that has been attached to the current state of the scenario.
Before you can add an action you must of course first select the event to which you want to assign the action. You can do this by tapping somewhere on the event symbol. Select then the requested action either by tapping the respective button on the toolbar or tapping the menu button on the event and selecting the action in the flyout menu.

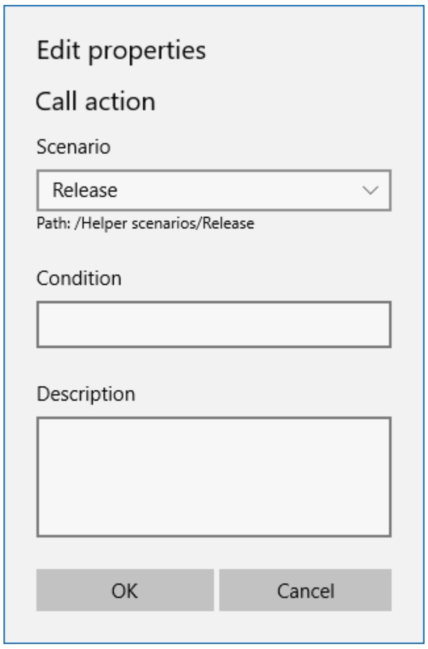


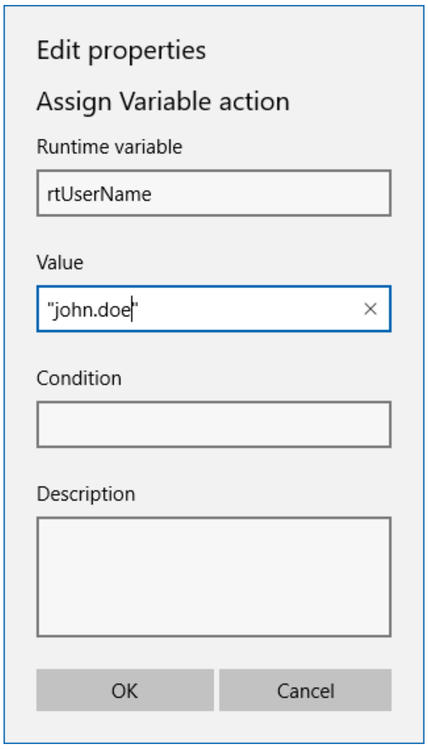
The following actions can be created:

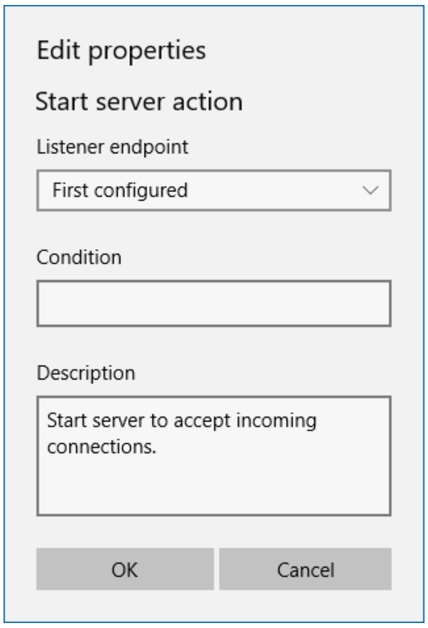
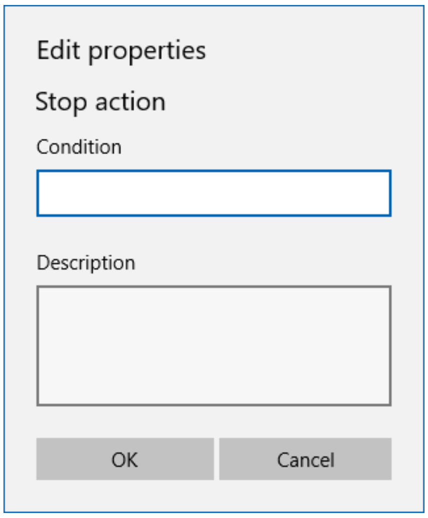
Action	Description	Action properties
Send message	Select this action when you want to respond to the event by sending out a message. You will be prompted to indicate which message you want to send out, define a possible condition and target address, and you can add some description. Message assignment To select your message tap the dropdown box. It allows you to pick any of the already registered messages but also allows you to create the message at this point in time if that hasn't been done yet. Condition You can provide a TestExpert expression here. If you do so then the action will only be executed when the expression evaluates to true or returns a non-zero numeric value. Example: `@var(RcvdToken) == 0x12` Target endpoint Tap the dropdown list to choose to which endpoint you want to send the message. You have 2 options: 1. When you select *Default endpoint* then TestExpert will send the message through what is called the default communication channel. Which channel that is will be depending on how the action is triggered: • When the action is triggered by a *Start* or *Timeout* event then this is the channel which TestExpert will create/reuse for the default target endpoint which you have to select before starting the scenario. • When the action is triggered by an *Incoming message* event then the channel is the one through which the incoming message was received. 2. When you select a specific target endpoint then TestExpert will create/reuse the connection towards that specific endpoint and use that one to send the message out. The endpoints listed in the dropdown are normally configured separately by opening the Solution Properties view but you can also	 Edit properties Send Message action Message: GetTemperature Path: /Basic messages/CoAp/GetTemperature Condition: Target endpoint: Default endpoint Description: This action sends a message to get the temperature of the sensor resource called "/s/t". OK Cancel  Target endpoint New endpoint... Default endpoint tcp://127.0.0.1:8000 tcp://192.168.0.241:8001 tcp://192.168.0.241:8002 tcp://192.168.0.241:8000

	add a new endpoint here by selecting *New endpoint…* in the dropdown list. Description Optional.	
Start timer	With this action you can request TestExpert to start a one-shot timer. You will be prompted to define the timeout value, a possible condition and some description. Timeout expression You can use a TestExpert expression to define the timeout value (in milliseconds.) Examples: 1000 `(@var(Step) * 2) + 1000` Condition See above. Description Optional.	 Edit properties Start Timer action Timeout expression (msec): 1000 Condition: Description:
Stop timer	This action stops any running timer. When selecting this action you will be prompted to define a possible condition and some description. Condition See above. Description Optional.	 Edit properties Stop timer action Condition: @var(MsgName) == "Stop" Description:

Set state	This action modifies the state of the scenario. You will be asked to define the new state, a possible condition and some description. State selection Select the new state by tapping the *State* dropdown box and tapping on the requested state. If the state isn't defined yet then you can also choose to create one here. Condition See above. Description Optional.	 Edit properties Set State action State: WaitAck Condition Description
Prompt	The *Prompt* action tells TestExpert to pause execution of the scenario and present a prompt text on the screen. You will be asked to enter the prompt text, a possible condition and an optional action description. Prompt Enter the text you want to show when the action triggers. Condition See above. Description Optional.	 Edit properties Give Prompt action Prompt message: The target system has issued an ACK message on the GetTemperature request. Condition Description

Call	The *Call* action allows you to interrupt the current scenario and execute another. When the called scenario terminates, TestExpert continuous with the original scenario where it left. You will be asked to enter the name of the scenario, a possible condition and an optional action description. Scenario Select the scenario that has to be called by tapping the dropdown box. Pick any of the already registered scenarios there. Condition See above. Description Optional.	 Edit properties Call action Scenario: Release Path: /Helper scenarios/Release Condition: Description:

Assign	With the *Assign* action TestExpert can be told to assign a value to a user defined variable. The variable will be created (if it doesn't exist yet) in TestExpert's runtime variables list.
	You will be asked to enter the name of the variable, an assignment expression, a possible condition and an optional description.
	
	Runtime variable
	
	You can assign any name you want to the variable. It is only when the scenario is actually running and the action is being triggered that TestExpert will either create the variable as a new run-time variable or use an already previously created run-time variable with the same name.
	
	Value
	
	You can use a TestExpert expression to specify the value to be set.
	Examples:
	`“john.doe”`
	`(@var(Step) * 2) + 1000`
	
	Condition
	
	See above.
	
	Description
	
	Optional.
		 Edit properties
		Assign Variable action
		Runtime variable: rtUserName
		Value: "john.doe"
		Condition:
		Description:

Start server	This action can be used to start a server service in TestExpert. You can trigger the action with any event in any state of the scenario but you will need it for sure with a *Start* event in state *Idle* when TestExpert must act as a server when the scenario runs. Listener endpoint Tap the dropdown list to select either the *First configured* listener endpoint or a specific endpoint. The endpoints listed in the dropdown are normally configured separately by opening the Solution Properties view but you can also add a new endpoint here by selecting *New endpoint…* in the dropdown list. Condition See above. Description Optional.	 Edit properties Start server action Listener endpoint: First configured Condition: Description: Start server to accept incoming connections.
Stop	The *Stop* action instructs TestExpert to stop executing the scenario. You will be asked to enter a possible condition and an optional action description. Condition See above. Description Optional.	 Edit properties Stop action Condition: Description:

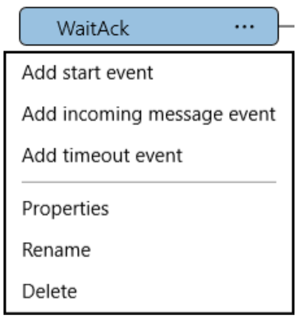
You can assign multiple actions to the event by repeating the above process. Any new action is automatically added after the last action. If you want to move an action before or after another one, then you can do this by selecting the action (by clicking on it) and dragging the action rectangle to the wanted position.

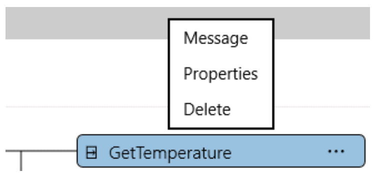
Action properties editor

As with states and events you can also here modify the action properties at any time by means of an *Action properties editor*. The editor is automatically displayed on the right side of the window when an action is selected and the view window is wide enough. If the windows is too narrow then tap the *More* menu button or right click on the action rectangle and select *Properties* in the flyout menu.

Additional actions on scenario items

Besides the above 'Add' actions (add State, Event, Action) there are some additional operations you can do for scenario items. You can reveal these commands either by tapping on the menu button in the item rectangle (you must first select the item) or by doing a right-click on the item.





Command	Description
Delete	Use this command to delete a scenario item (state, event, or action). Of course, deleting a state or an event will also remove the attached events or actions. Note: Be careful since there is no un-delete possibility.
Properties	Use the *Properties* command to open the item's *Properties editor* in a separate view page.
Rename	The *Rename* command is only supported for a *State* item.
Message	The *Message* command is only supported for *Incoming message* events and *Send message* actions. It allows you to open a tab view to look at or modify the message content.
Cut, Copy and Paste	Not supported yet.

Import an already existing scenario

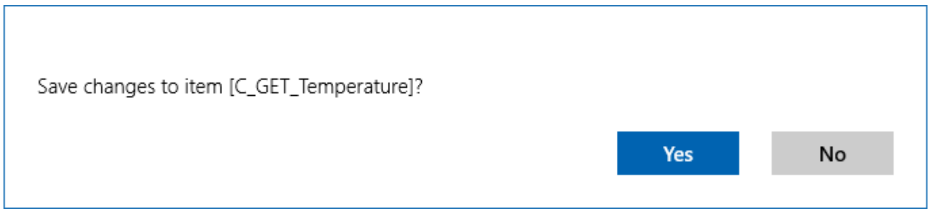
The «Import» button at the top of the scenario editor lets you import a complete scenario that already exists in the solution file.

⟻
Import

All definitions that already existed prior to the import will be cleared and all definitions of the scenario that is being imported will be copied.

Saving the scenario

You must be aware that any change you make while editing is done on a temporary copy of the original scenario. In order to make the changes persistent you should therefore save the updated scenario.
You can do that by tapping the *Save* button in the navigation menu or by tapping the *Yes* button in the dialog that comes up when you close a scenario that has unsaved changes.


Save changes to item [C_GET_Temperature]?

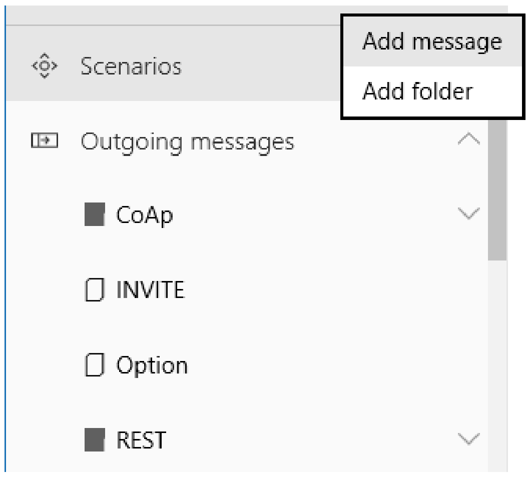
In both cases the update in the solution file is done for all items in the tabs area that have unsaved changes.

Managing messages

This chapter explains how TestExpert lets you add and adapt messages in your solution. Because scenarios have to treat messages differently depending on whether they are outgoing or incoming you will notice that the navigation panel comes with an Outgoing messages group and an Incoming messages group.
In case however you don't use scenario's and simply want to create and send messages out to a target device by yourself then the Outgoing message group is the place where you have to define and manage your messages.

Create a message

To create a new message select the *Outgoing messages* or *Incoming messages* group menu item in the navigation panel or a subfolder in it.
Tap on the *More options* icon to the right of the selected item.
Alternatively you can also right-click on the group menu item or on the folder.



Select *Add message* on the flyout menu.

TestExpert then adds a new message to the solution with a default name. Change the name into something meaningful.

You can then define the message content using the Message editor that is added by means of a tab item in the tabs area.

Note: As with scenarios it is also possible to create a new message in a folder (or subfolder). To do that you must first create the folder/subfolder by selecting *Add folder* on the main *Outgoing messages* or *Incoming messages* menu item or on an already existing folder item.

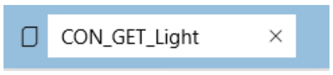
Modify a message

Select the message in the navigation panel.

Modify the message using the Message editor.

Rename a message

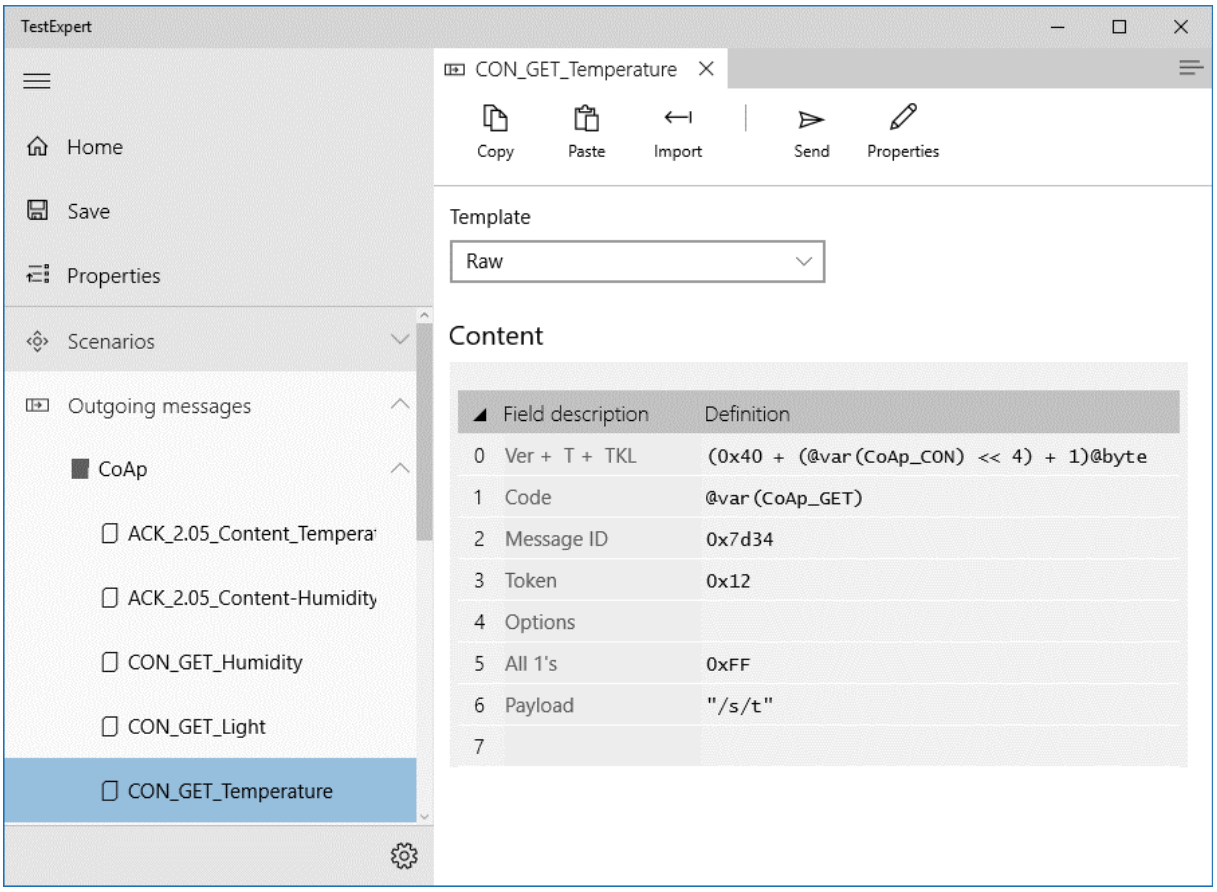
You can rename a message in a couple of ways:

- Tap twice on the message in the navigation panel. A small edit box is shown where you can change the name.
  

- Select the message in the navigation panel and tap on the *More options* icon to the right of the message.
  Select *Rename*.
- When you have the Message properties editor open you can also rename the message by simply changing the name in the upper *Name* edit box.

Message editor

The Message editor provides a view that allows you to define the content of a message.



Template: Raw

Content

	Field description	Definition
0	Ver + T + TKL	(0x40 + (@var(CoAp_CON) << 4) + 1)@byte
1	Code	@var(CoAp_GET)
2	Message ID	0x7d34
3	Token	0x12
4	Options	
5	All 1's	0xFF
6	Payload	"/s/t"
7		

It consist of a number of parts:

1. A top bar holds a number of action buttons that allow you to do things like: copy, paste, import, send the message, and look at some message properties.
2. Right under the command bar there is *Template* dropdown box that lets you choose how you want to define the content of the message. Depending on which template you select a specific editor will come up. You can choose from:
   - *Raw* – This is a built-in template that gives you full freedom with respect to what kind of data has to go into the message. The message editor allows you to enter the data as: one or more byte values, word, integer, long, string, or a combination of those. You can also make use of expressions supporting arithmetic and logical operators to define more complex data structures.
   - *HttpMessage* – This template lets you define both HTTP request and response messages. A powerfull HTTP message editor is available which allows you to define all the HTTP message properties (method, status code, headers, body, etc.) in a user-friendly way.
   - *An ABNF template file* – ABNF template files are text files which describe the structure and format of the messages you want to deal with. The files use a well-known formal message syntax notation that is specified by the IETF in RFC5234 (*Augmented BNF for Syntax Specifications*). ABNF lends itself

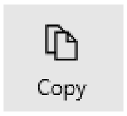
primarily for text-based message protocols.
TestExpert comes with an ABNF parser and editor which presents a given message layout and possible editable parts in a user-friendly way.
Note: If you want to make use of ABNF template files to describe a message you must add them first to your solution by opening the Solution properties page.

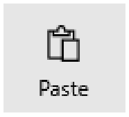
3. The biggest part on the screen is taken up by the actual editor. There are current 3 type of editors:
    - Raw message editor
    - HTTP message editor
    - ABNF message editor

Note: When the window/display is wide enough then TestExpert also brings up the Message properties editor on the right side. The *Properties* action button on the top command bar is then hidden.

Copy and paste action

You can easily copy the content of a message (or a message element) into another message (or message element) by using the *Copy* and *Paste* command buttons.

- Open the message (or element) you want to copy and tap or click the Copy button.
  

  This puts the content in the clipboard.
- Then, open the message that needs to receive the copy and tap or click the Paste button.
  

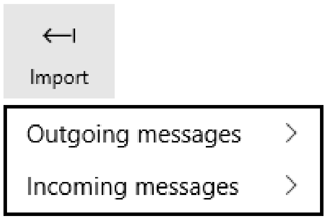
Notice that the action completely overwrites the content (if any) with the source data.

Copy/paste is allowed between any type of message (e.g. copy outgoing message data to an incoming message) but not between a message and a message element. After having received the copy the template property of the message is set to the one of the source message.

Since the copy is in the clipboard you can repeat the paste action on other messages.

Import action

The import action also lets you copy the content of any other message (or message element) into the message that is being edited. This is different from copy/paste since it doesn't use the clipboard but instead asks you to select the source message (or message element) from a list.

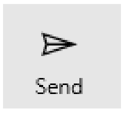


Notice also here that the action completely overwrites the content (if any) with the source data and that, in case of a message, the template property is set to the one of the imported message.

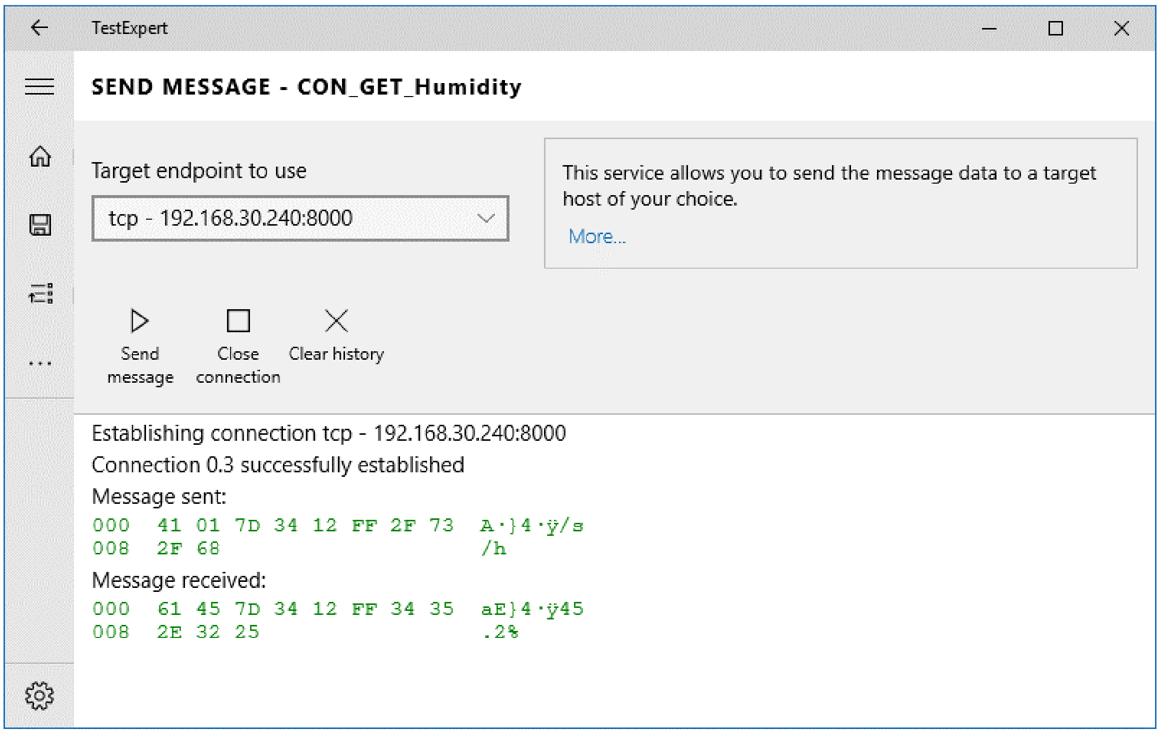
Send action

It is possible to send the message data to one of the endpoints that has been configured as a target endpoint for the solution.

If you want to do that tap or click the *Send* button.



This opens the *Send Message* view on top of the tabs area where you can select the target endpoint/host and request TestExpert to send the message.



SEND MESSAGE - CON_GET_Humidity

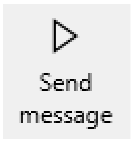
Target endpoint to use: tcp - 192.168.30.240:8000

This service allows you to send the message data to a target host of your choice.

```
Establishing connection tcp - 192.168.30.240:8000
Connection 0.3 successfully established
Message sent:
000  41 01 7D 34 12 FF 2F 73  A·}4·ÿ/s
008  2F 68                     /h
Message received:
000  61 45 7D 34 12 FF 34 35   aE}4·ÿ45
008  2E 32 25                  .2%
```

The *Target endpoint to use* dropdown box lets you select a target endpoint. As a default TestExpert will preselect the first configured endpoint. Target endpoints must be configured in the Solution properties view.

Tap or click the *Send message* button to send the message out.



The system will then establish a connection to the selected endpoint (if it doesn't exist yet), format the message definition into a set of bytes, and then actually send the message.

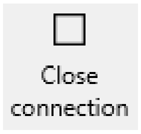
Note: the encoding of the message payload is depending on the type of message:

- With a Raw message the encoder uses the endianness and string encoding format (i.e. UTF8) that is configured in Solution properties.
- With a HTTP and ABNF type of message the text is encoded using UTF8.

After the message is sent TestExpert itself will not close the connection for the following reasons:

- By keeping the connection open the system can show you possible messages that are coming back (e.g. as a response) from the target system.
- It allows you to send successive messages without the overhead of always building up a new connection.

If you want you can force a connection that is active for the selected target endpoint to close by tapping the *Close connection* button.



To clear the history in the report window tap the *Clear history* button.

Properties action

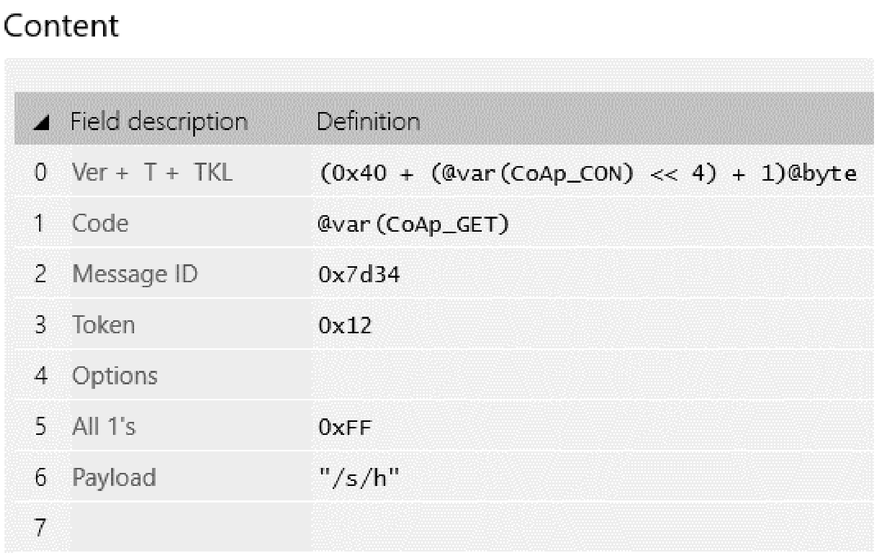
The *Properties* action button is only shown when TestExpert runs in a small window. Tap or click on it to popup the Message properties editor.

Raw message editor

The *Raw message editor* allows you to define messages at the lowest possible octet level. The editor lets you define the content of all type of messages: messages that consist of just a set of octets, messages that contain strings that are delimited in some way, or messages that contain both octets and strings.

The editor provides you with a view which presents a message table consisting of 3 columns.

Example:



Content

	◢ Field description	Definition
0	Ver + T + TKL	`(0x40 + (@var(CoAp_CON) << 4) + 1)@byte`
1	Code	`@var(CoAp_GET)`
2	Message ID	`0x7d34`
3	Token	`0x12`
4	Options	
5	All 1's	`0xFF`
6	Payload	`"/s/h"`
7		

The purpose of the message table is to present the message content as a set of protocol data elements (octets, strings, ..). Although it is theoretically possible to define the whole message using a single row of data, this is not what you should do. Indeed, multiple rows can be created which allow you to specify the message as consisting of a set of different information elements. There is no limit in TestExpert on the number of rows that you can use.

The columns:

- The first column simply contains the row number.
- The second column of each row, called *Field description*, doesn't need to have content. It can be used to clarify the meaning of a row/definition for the user.
- The third column, called *Definition*, is used to hold the definitions of the actual message information elements.
  In the case of an incoming message it can also contain assignment statements to save received data values in runtime variables.

When the message is initially created a single empty row is added to the table.

Defining content

The content of a message must be defined by entering values in the cell(s) of the definition column. To do this TestExpert comes with a *Message description language* that allows you to enter the content data in various ways. The next couple of examples should give you an idea. For a full description of all possibilities you should read the reference text in the Message description language chapter.

Example 1 - Using simple string and value statements.

Assume the following content in the table:

Field description	Definition
A string	`"--begin--"`
A byte	`1`
A byte and a word	`0x20 0x0010`
A string	`"--end--"`

The resulting message content will be:

```
Content:

000   2D 2D 62 65 67 69 6E 2D   --begin-
008   2D 01 20 00 10 2D 2D 65   -. ..--e
010   6E 64 2D 2D               nd--
```

Note that byte ordering will be important when using the *Raw message editor*. In the above example the 0x0010 value will be sent out as 0x00 followed by 0x10. You can however change the ordering for word, integer and long values by setting the *Endianness* property in the Solution properties page to little endian or big endian.

Example 2 - Using value modifiers and expressions.

Assume the following content:

Field description	Definition
A word (16 bits)	`1w`
An expression	`(0x20 + 0x02)`
Conversion to an integer (32 bits)	`(0x20 + 0x02)@int`

The resulting message content will be:

```
Content:

000   00 01 22 00 00 00 22      .."..."
```

Example 3 – Using variables

In addition to scenarios and messages TestExpert also allows you to define one or more variables. They can help you when you have to specify certain, more or less fixed values in multiple messages or when you want to include a value that is extracted from the data content of a received message.
A variable consists of a name and a value and TestExpert allows you to reference them in the various *Message editors*.

Assume that the solution contains 4 variables ‘cr’, ‘lf’, ‘crlf’, and ‘block’ with the following values assigned to them:

Variable name	**Definition**
cr	`0x0D`
lf	`0x0A`
crlf	`@var(cr) @var(lf)`
block	`10 (10 + @var(lf)) 30 "ABC"`

Assume next that the content definition of a message looks as follows:

Field description	**Definition**
Variable ‘cr’	`@var(cr)`
Variable ‘crlf’	`@var(crlf)`
Variable ‘block’	`@var(block)`

The resulting message content will then be:

```
Content:

000  0D 0D 0A 0A 14 1E 41 42  ......AB
008  43                       C
```

Note: The use of variables is described in full detail in the chapter TestExpert variables.

Example 4 – Using Outgoing and Incoming message elements

When you define a message using the raw format template it might be convenient to make use of separate re-usable Message elements and include those as fields at a given place in the message. See Managing message elements for more details.

The next table shows the definition of a Message element called ‘UserName’.

:

Field description	**Definition**
Header	`"Name: "`

Name passed as parameter	`@this.parm(1)`
Field terminator	`@var(crlf)`

Note in the above table that the ‘UserName’ information element is not a constant field but that it assumes that the ‘caller’ will pass a parameter value holding the name.

The message in which we want to include the element then looks as follows:

Field description	**Definition**
Request	`"REGISTER" @var(crlf)`
UserName field	`@element("ome/UserName", "John.Doe")`
Message terminator	`@var(crlf)`

The ‘UserName’ field is included by making use of the *@element* function. The first argument of this function is the pathname of the element that must be included. The second argument is the first (and only) parameter expected by the element.

The resulting message content will then be:

```
Content:

000  52 45 47 49 53 54 45 52  REGISTER
008  0D 0A 4E 61 6D 65 3A 20  ..Name:
010  4A 6F 68 6E 2E 44 6F 65  John.Doe
018  0D 0A 0D 0A              ....
```

Editing the message table

Edit cell content

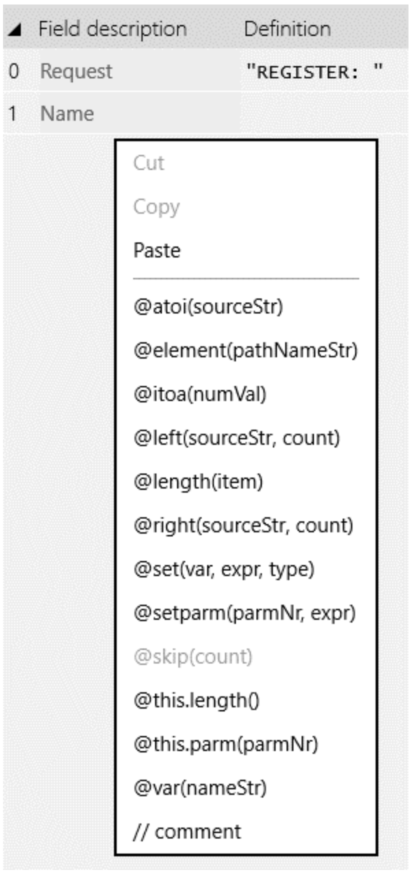
To enter/change text in any of the description and definition table cells tap or click in the cell. A text box comes up with a white background. It comes with a context menu with support for copying and pasting text, and (in a definition cell) for inserting Message description language statements.

The "clear all" button in the text box lets a user quickly delete all text that has been entered.

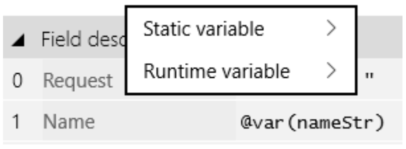
When a cell is in edit mode pressing the **Up/Down** navigation key will let you immediately edit the content of the cell in the previous/next row.

Pressing the **Tab** key will bring up the cell to the right of the current cell. When on the last cell of a row, press the **Tab** key twice to bring up the first cell on the next row.

To bring up the context menu right-click on the cell. The commands you will see are dependent on which cell you have clicked. For a ‘description’ cell the usual Cut, Copy, Paste and Select All commands are presented. For a ‘definition’ cell you will see extra commands to insert Message description language statements.



Tap or click on one of the statements to insert the statement in the cell. In case the chosen statement needs additional input parameters (like: a message, a variable name, etc.) you can again right-click on the parameter to get a new context menu that helps you with selecting the value of the parameter.



Inserting rows

To insert a new row before a given row, fully select the given row by tapping or clicking in the small row number cell. Then add the new row using one of the following options:

- Right-click and select *Insert* in the flyout menu.
- Press the **Ctrl** + **+** (plus sign) key or the **Ctrl** + **Shift** + **+** (plus sign) key.

In any case the new row is added right above the selected row.

Deleting rows

To delete a row, select the row to be deleted by tapping or clicking in the small row number cell. You can then delete the selected row using one of the following options:

- Right-click and select *Remove* in the flyout menu.
- Press the **Ctrl** + **-** (minus sign) key or the **Ctrl** + **Shift** + **-** (minus sign) key.

Cutting and copying rows

You can cut/copy rows and paste them in a given row (i.e. overwrite the row). There are 2 possibilities available to do that:

- Right-click on the row you want to cut/copy and select the *Cut* or *Copy* command.
  Then right-click the row that needs to get the content of the cut/copied row and select *Paste* in the flyout menu.
- Select the row you want to cut/copy and press the **Ctrl** + **X** / **Ctrl** + **C** key.
  Then select the row that must be overwritten and press the **Ctrl** + **V** key.

After you copied a row the original row remains 'selected-for-copy' meaning that you can repeat the paste command. To stop this, press the **Esc** key.

Raw incoming message validation

When an incoming message is defined using the *Raw message editor*, then TestExpert uses the definitions in the second column to compare with all the octets of the received message. The definition expressions that are used are there to specify different kind of matching conditions, such as: byte match, string match, skipping of bytes that doesn't have to be checked, checking for equality with variables, etc..

Given a message, received from the system under test, carrying the following bytes (in hex):

```
01 FE 80 20 30
```

You have different possibilities to define this message in your scenario.

The message content is completely known beforehand

With this method you completely define all the bits and bytes of the message. For the above example, the message could look as follows:

Field description	**Definition**
First byte	`0x01`
Next 2 bytes	`0xFE 0x80`
Fourth byte	`" "`
Last byte	`(0x28 + 0x08)`

Validating the above message against this expression results true because the individual bytes of the message correspond with the outcome of the various expressions.

The validating process roughly happens as follows:

- The current position pointer in the message is set to the first byte of the received message.
- The first value of the expression, i.e. 0x01, is compared with the message value at the current position of the field pointer
- The second statement in the expression, i.e. 0xFE 0x80, tells the system to check the next byte against 0xFE, move the current position pointer 1 up, check the next byte against 0x80 and again move the position pointer 1 up. We are now at the fourth byte.
- The third value in the expression, i.e. " " denotes a string. The string is compared with the message byte(s) starting at the current position of the pointer. Because the string only contains a single 'space' character, only 1 byte in the message (the one with value 0x20) is compared. The position pointer moves as much bytes to the right as given by the length of the string.
- The last statement involves an addition of 2 values. The result of this calculation is 0x30 which is compared with the value at the pointer position.

Message content is only partially known

Let’s assume that the incoming message, specified above, can carry different values in byte 2 and 3, depending on conditions which you cannot control. So you need a method to ignore these 2 bytes.
That’s exactly the purpose of the `@skip()` statement. You can use it as follows:

Field description	**Definition**
Expression	`0x01 @skip(2) " " (0x28 + 0x08)`

Only look at the first part of the message

You can tell TestExpert only to look at for instance the first byte and to ignore the remainder of the message by defining the `@skip(*)` statement. You use it as follows:

Field description	**Definition**
Expression	`0x01 @any`

Conditional interpretation of message data

If your protocol messages carry information elements of which the format and content is dependent on certain bits or bytes in the message, then you might have to use TestExpert’s conditional expressions to check them.

Consider an information element which carries a single byte length value in the 2nd octet if bit 8 in the first octet is set to 1, and a 2 byte length value in the 2nd and 3rd octet if the bit is not set.

- In case of a single byte length you want the first octet to be 0x80 and the length to be 0x01.
- In case of a 2 byte length you want the first octet to be 0x7F and the length value to be 0x0005. You can use TestExpert's `@if` and `@$` statements to account for this.

You will have to code that part of the message as follows:

Field description	Definition
Octet 1, 2 (and 3)	`@if(@$ & 0x80) ? {0x80 0x01} : {0x7F 0x0005}`

Explanation

`@$` in the `@if()` statement refers to the value of the byte at the current parser position, i.e. the first octet.

If the `@if()` expression is true then, starting from the current parser position the next 2 bytes must be 0x80, 0x01.
If the expression is false, then at the current parser position there must be a 0x7F byte followed by 0x00, 0x05.

Raw incoming message extraction

Assume a message, holding 3 octets in octet 2, 3 and 4 which are unknown upfront.

When the message is received in a scenario and matches the current scenario state octets 3 and 4 need to be saved in a run-time variable in order to use them later in the scenario in one of the outgoing messages.

To store these 2 octets, you will define the message as follows:

Field description	Definition
First byte	`0x01`
3 unknown octets	`@skip(1) @set(rcvdRef, @$(2), @word) @skip(2)`
Fourth byte	`" "`
Last byte	`(0x28 + 0x08)`

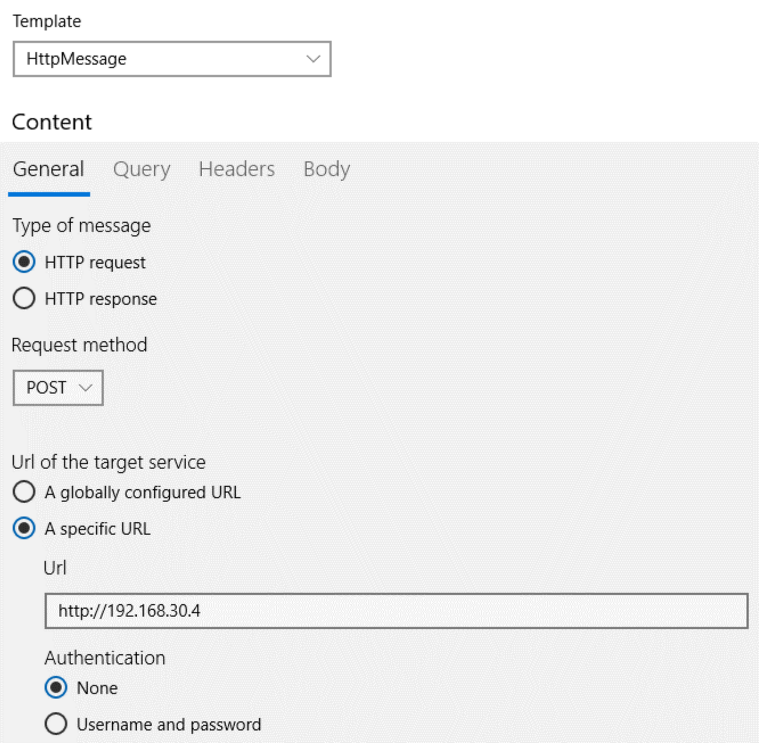
The `@skip(1)` statement in the definition column of the 2nd row skips the first octet of the second row (which is octet 2 in the message).

The `@set(rcvdRef, @$(2), @word)` command then takes the next 2 octets at the current parser position of the received message, converts them to a word value (16 bits) and stores the value in the runtime variable 'rcvdRef'.
Because a `@set()` statement doesn't move the parser position, there is still a `@skip(2)` statement necessary to skip these 2 bytes and move the parser position up.

HTTP message editor

The *HTTP Message Editor* is the 2nd type of editor in TestExpert. It lets you define HTTP request messages (GET, POST, etc.) and HTTP response messages (200 OK, 401 Unauthorized, etc.) in a very easy way.

To invoke the editor click *HttpMessage* in the *Template* selection dropdown box that is presented above the *Content* section of any message editor.


Template: HttpMessage

Content

General | Query | Headers | Body

Type of message
- [✓ HTTP request]
- HTTP response

Request method: POST

Url of the target service
- A globally configured URL
- [✓ A specific URL]

Url: http://192.168.30.4

Authentication
- [✓ None]
- Username and password

The editor groups the various properties of a HTTP message in the following groups:

- General
- Query
- Header, and
- Body.

General

In the *General* group you have to select the type of message: a request or a response.

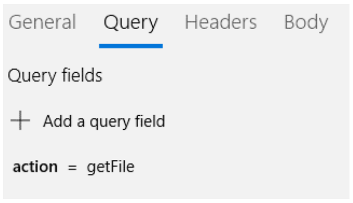
- With a request message you additionally have to select:
  - The request *Method* (GET, PUT, ..),
  - An optional *URL*, and
  - Optional *Authentication parameters*.

    When you select *'a globally configured URL'* then TestExpert will not assign a URL to the message but it will let you select the URL later on when the message is used in a scenario or when you want to send it out. This allows a given message to be sent easily to multiple devices.

- With a response message you only have to select the *Status Code* (200 OK, ..) in this group.

Query

The *Query* group is available in case of a HTTP request message.

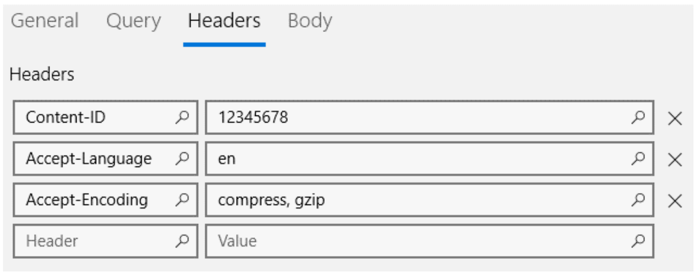


action = getFile

When you add or modify a query field a dialog pops up that lets you specify the field's name and value.

Header

The *Header* group is the place where you define general, non-content related message headers.



Content-ID	12345678
Accept-Language	en
Accept-Encoding	compress, gzip
Header	Value

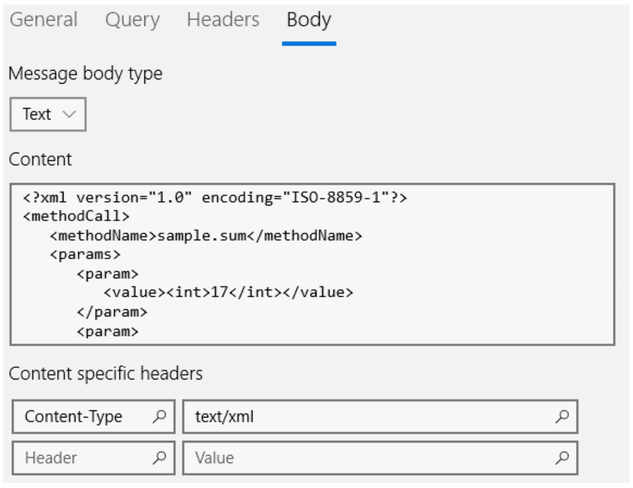
To add a header, tap the empty *Header* box in the last header row and type or select the header name from a list of possible header names. Then tap the *Value* box on the same row and type or select the value.
To delete a header, tap the X button shown on the right side of the corresponding row.

Body

The *Body* group lets you define the content of the body part of the HTTP message and content-specific HTTP headers (like Content-Type, Content-Id, ..). You can define the content as being: text, a file, url-encoded form data, and multipart data.

Text body

When you select *Text* as content for the *Body* group you will be given the possibility to enter the text (in the *Content* textbox) and additionally provide content-related headers that apply to the content you enter.



Message body type

Text

Content

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<methodCall>
   <methodName>sample.sum</methodName>
   <params>
      <param>
         <value><int>17</int></value>
      </param>
      <param>
```

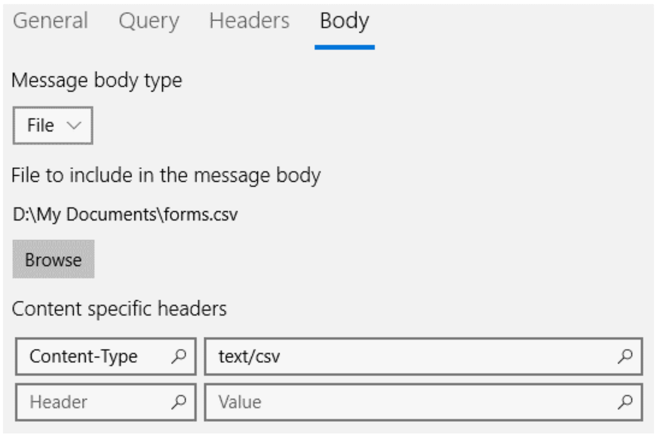
Content specific headers

Content-Type	text/xml
Header	Value

Note that TestExpert will force you to provide a value for the *Content-Type* header that it automatically will add to the *Body*. This header cannot be deleted.

File body

Selecting *File* tells Expert to take the content of a given file and use that as content for the message body.



Message body type

File

File to include in the message body

D:\My Documents\forms.csv

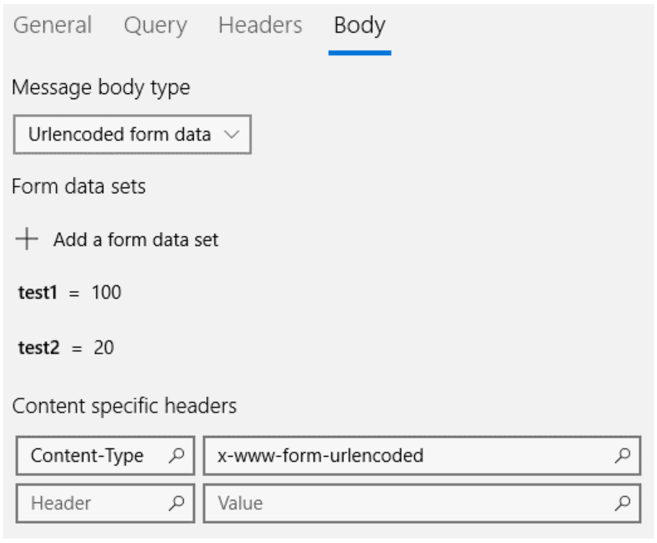
Content specific headers

Content-Type	text/csv
Header	Value

You must select the file by tapping the *Browse* button and, same as with a *Text* body, you can define one or more content headers (of which the *Content-Type* header is mandatory.

Url-encoded form data

Select *Urlencoded form data* in case you want to have a message body with form data fields that have to be url-encoded.



Message body type: Urlencoded form data

Form data sets

test1 = 100

test2 = 20

Content specific headers

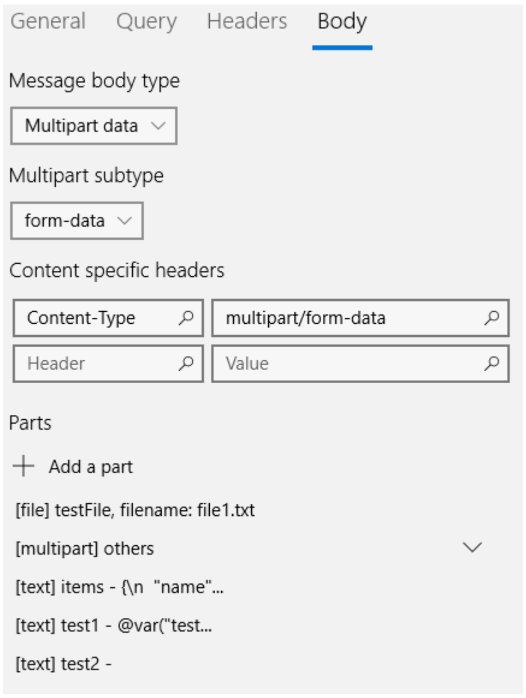
Content-Type	x-www-form-urlencoded
Header	Value

To add a form data set tap the *Add a form data set* button. This bring up a dialog where you can provide a name and a value for the set.
As with the other type of content bodies you can also add content specific headers.

Important: Don’t change the automatically added *Content-Type: x-www-form-urlencoded* header!

Multipart body

Select *Multipart data* when you want to have a message body that contains multiple entities of data.



Message body type: Multipart data

Multipart subtype: form-data

Content specific headers

Content-Type	multipart/form-data
Header	Value

Parts

[file] testFile, filename: file1.txt

[multipart] others

[text] items - {\n "name"...

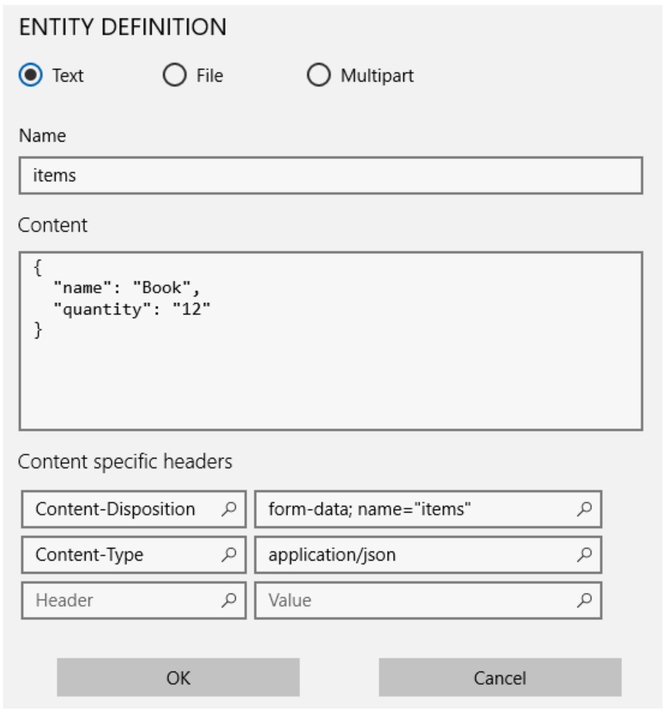
[text] test1 - @var("test...

[text] test2 -

You must then also indicate what the purpose is of those entities by selecting one of the options in the *Multipart subtype* dropdown box. You can choose: *form-data*, *mixed* data, *alternative* data, and *related* data.

Important: Don’t change the *Content-Type: multipart/..* header that is automatically added.

To add the various parts tap the *Add a part* button. This pops up a dialog where you can specify the content of the multipart entity.



Name: items

Content:

```
{
  "name": "Book",
  "quantity": "12"
}
```

Content-Disposition	form-data; name="items"
Content-Type	application/json

You have to define the following properties:

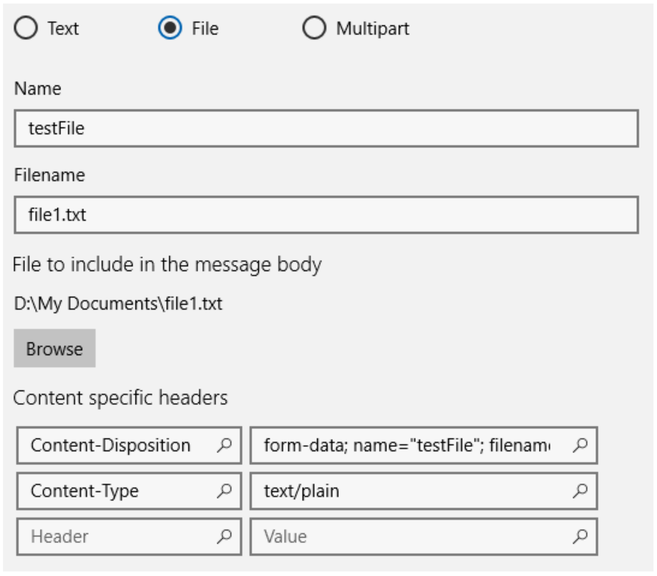
- The type of content - It can be: *Text*, a *File*, or *Multipart* data. TestExpert will automatically add a default *Content-Type* header. You are free to change the value of this header except when selecting *Multipart*. In that case the type is set to the selected multipart subtype (multipart/mixed, multipart/alternative, or multipart/related).
- A possible name for the entity – The *Name* textbox is only available when the entity is a part of a multipart/form-data body. In that case TestExpert will also automatically add a *Content-Disposition* header to the set of content headers. The value of this header is filled in by TestExpert (using the value from the *Name* textbox).
- Possible content related headers – Add these headers by tapping the *Header* textbox in the last row of the headers list and typing or selecting the header name from a dropdown list. Do the same with the *Value* textbox.
- The only thing left to do then is to define the content itself. How to do that is depending on the type of content.

Text Content

The text itself must be defined in the *Content* textbox.

File Content

In case the entity is part of a multipart/form-data body you must define the name of the file in the *Filename* textbox. It is used (together with the *Name* entry) by TestExpert to set the value of the *Content-Disposition* header.



Text [✓ File] Multipart

Name: testFile

Filename: file1.txt

File to include in the message body

D:\My Documents\file1.txt

Browse

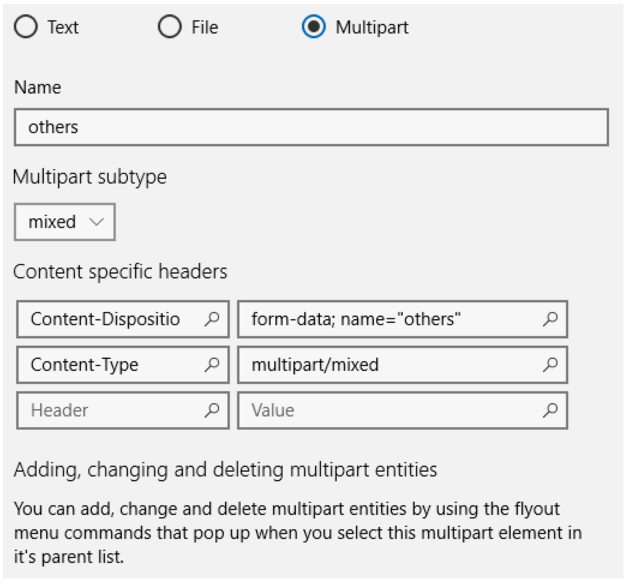
Content specific headers

Content-Disposition	form-data; name="testFile"; filenam
Content-Type	text/plain
Header	Value

The file itself is selected by tapping the *Browse* button.

Multipart Content

You must select the multipart subtype from the *Multipart subtype* dropdown box. You can select: *mixed*, *alternative*, and *related*. TestExpert will set the part's Content-Type header accordingly.



Text File [✓ Multipart]

Name: others

Multipart subtype: mixed

Content specific headers

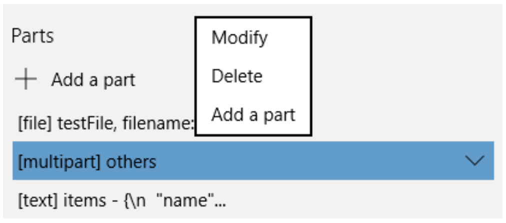
Content-Dispositio	form-data; name="others"
Content-Type	multipart/mixed
Header	Value

Adding, changing and deleting multipart entities

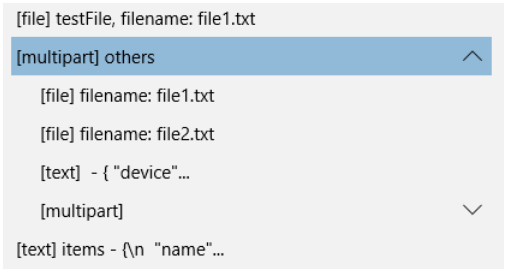
You can add, change and delete multipart entities by using the flyout menu commands that pop up when you select this multipart element in it's parent list.

As the explanation in the dialog says you have to close the dialog and select the multipart entity in the *Body* part of the message when you want to manage the entities which you want to have inside this multipart element.

To add a sub-part right-click the parent multipart and select *Add a part* from the flyout menu.



To reveal the configured sub-parts simply tap the multipart entity in the list.


[file] testFile, filename: file1.txt
[multipart] others
[file] filename: file1.txt
[file] filename: file2.txt
[text] - { "device"...
[multipart]
[text] items - {\n "name"...

To hide the sub-parts tap again the parent multipart entity.

To edit a sub-part right-click on the sub-part and select *Modify* from the flyout menu.

Hard-coded versus MDL-driven HTTP messages

As with all other message editors in TestExpert you can define the payload values for most of the HTTP message elements in outgoing and incoming messages either as hard-coded values or as a combination of hard-coded values and values that are resolved using TestExpert’s Message Description Language (MDL).

An example of the latter method is the use of the @var() expression to get the value from a runtime or static variable. This is useful when you want to work with messages where the payload has to be different in some circumstances or is depending on other messages that have been received/sent in a test scenario.

Another example is the definition of an incoming message. It will often be very hard to upfront describe an incoming message in full detail: e.g. because the remote system doesn’t react in a static way and the message content is therefore not entirely predictable. You often are also not interested in all of the message elements.
To help you with this TestExpert provides the MDL @match() and @skip() expressions.

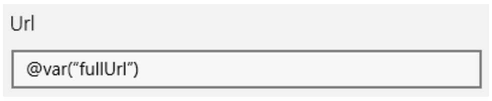
The @match() statement allows you to validate a particular element of an incomng message using a regular expression (i.e. without fully defining the actual strings).

The @skip() statement lets you tell Expert to skip the validation for a number of bytes or for the full element.

The use of the MDL is supported for the following HTTP message elements:

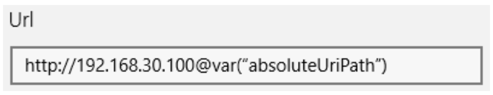
- URL
- Query field names and field values
- Header field values
- Text body values

Example 1 - Assume you have a variable *fullUrl* set to “`http://192.168.30.4`”. You can then define the URL in a HTTP request message as:


Url

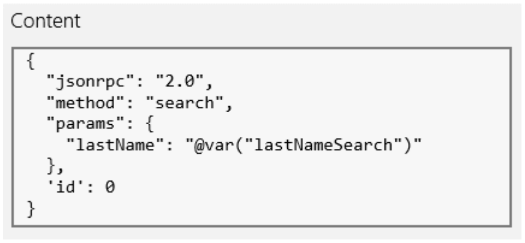
@var(“fullUrl”)

Example 2 - You can also mix hard-coded values with variable data


Url

http://192.168.30.100@var(“absoluteUriPath”)

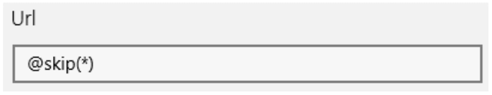
Example 3 – A similar example but now for the text content in the body part of a message.


Content

```
{
  "jsonrpc": "2.0",
  "method": "search",
  "params": {
    "lastName": "@var("lastNameSearch")"
  },
  'id': 0
}
```

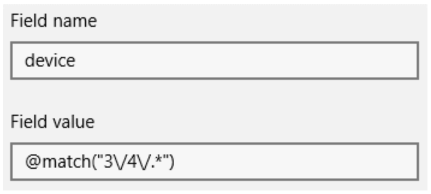
The json object definition in the above example refers to a TestExpert variable *lastNameSearch*. When sending the message out TestExpert will substitute the expression with the actual value of the variable.

Example 4 - Use the @skip(*) statement to ignore the full value of an element of an incoming HTTP message. In the example the URL path is ignored.


Url

@skip(*)

Example 5 - Assume that the field value of a given query field in an incoming request message contains a path-like string (e.g. “3/4/100”) and we are not interested in the value of the last part.


Field name

device

Field value

@match("3\/4\/.*")

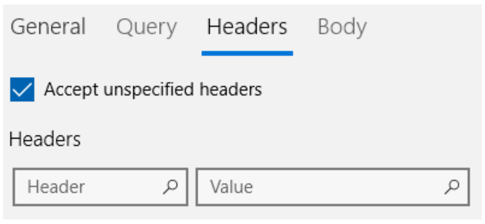
This matches: “3/4/100”, “3/4/7”, etc, but not “4/4/100”. TestExpert will process the @match() statement by trying to find a match in the incoming message using a regex where the regex pattern is «3\/4\/.*».

Note - Because regular expressions are an important tool in TestExpert when dealing with message validation and also information extraction (see later) for HTTP messages you will need some knowledge about it.
A good information source for this is Wikipedia.
A quick reference of the *Regular Expression Language* (as supported by the .NET engine in TestExpert) can be found here and here.

A good website to verify your regex patterns against content of your choice is .NET Regex Tester - Regex Storm.

Ignore unspecified query fields and headers in incoming HTTP messages

With incoming HTTP messages the editor gives you the possibility to ignore unspecified query fields, general header fields, and header fields in the body part of a message. That option is set as a default.



☑ Accept unspecified headers

Headers

Header	Value

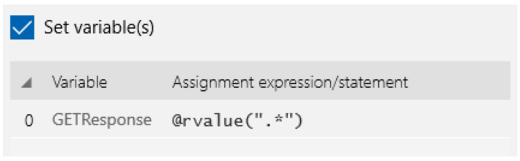
When you unselect the option then TestExpert will validate all received fields against the definitions that it finds in a given scenario message. If it fails to find a matching field then the message will be invalid.

Assign values to variables while executing a scenario

When a scenario is executing and an event comes in that matches with a given incoming message event then TestExpert allows you to assign values to runtime variables in 2 ways:

- By having specific Assign Variable actions attached to the matching event.
- By having assignment statements in the event's incoming message specification. The HTTP elements for which this is supported in the editor are:
    - URL (HTTP server),
    - Query fields (HTTP server), and
    - Text body (HTTP client and server)

The assignment statements have to be entered in a table that looks as follows:



☑ Set variable(s)

	Variable	Assignment expression/statement
0	GETResponse	@rvalue(".*")

You can add as many rows as you like. Managing rows (add, delete, copy, etc.) is the same as with the Raw message editor.

The left cell in each row holds the variable's name.The right cell contains a statement that provides the value to be stored in the variable.

These are the possibilities:

Action	Content of the Assignment expression/statement cell
Set a hard coded numeric value, e.g. 100.	100
Set a hard coded string value, e.g. "abc".	"abc"

Set a value using an expression with a @var statement.	`(1 + @var(id))`
Extract a value from the HTTP element using a regular expression and assign it to a variable.	`@rvalue(“pattern”)`
Extract a value from the HTTP element using a JSONPATH expression and assign it to a variable.	`@jvalue(“jsonpath”)`
Extract a value from the HTTP element using an XPATH expression and assign it to a variable.	`@xvalue(“xpath”)`

Regex example

Store all characters of a string (up to a newline character) in a variable

```
@rvalue(".*")
```

HTTP incoming message validation

TestExpert validates HTTP messages received in a client or server test scenario as follows:

HTTP response messages (client scenario)

- The status code in the HTTP status line must match the one defined for the message in a scenario’s Message Event.
- The reason phrase in the HTTP status line is completely ignored.

HTTP request messages (server scenario)

- The request method must match the one defined for the message in a scenario’s Message Event.
- The absolute URL path must match with the definition.
- All query fields defined in the solution message must be present in the received message.

HTTP request and response messages (client and server scenario)

- All the general header fields defined in the solution message must be present in the received message.
- All the content related header fields specified in the solution message must be present.
- Finally the ‘content’ part defined in the solution message is checked against the content in the received message.

Automatically included headers when sending

Some headers, when not defined by the user, will automatically be included by TestExpert when it sends the message in a scenario or in the Send Message view.

These headers are:

- For HTTP Request messages:
    - Accept-Encoding: gzip, deflate
    - Host: <set to the target address>
    - Connection: Keep-Alive
    - If-Modified-Since – Only with GET requests.
    - Content-Length – With request messages carrying a body element. The length is set to the size of the included body.
- For HTTP Response messages:
    - Server: TestExpert/1.0
    - Content-Length: <size of the included body>

ABNF message editor

The ABNF message editor is the 3th type of message editor in TestExpert.

If you are dealing with text based protocol messages then TestExpert gives you the possibility to define the content of those messages in a very simple way using ABNF template files. TestExpert has a built-in parser for such files and its *Template message editor* can present single message elements, multiple elements and optional elements in a user-friendly way.

For example consider the following (simple) ABNF file supporting 2 type of messages:

- 3 type of request messages: SwitchOn, SwitchOff, and Reset. The first 2 carry a parameter which defines the device ID.
- 2 response messages: OK and NOK.

The ABNF file defining the rules for this protocol would look as follows:

```
; !name("DeviceProtocol device-easy.abnf")
; !syntax("abnf")
; !import("core-abnf.abnf")
; !import("ALPHA","CRLF","DIGIT","DQUOTE","HEXDIG","HTAB","OCTET","SP","WSP")

message = request / response
request = switch-on / switch-off / "Reset"
response = "OK" / "NOK"
switch-on = "SwitchOn" ":" device-id
switch-off = "SwitchOff" ":"  device-id
device-id = <?>          ; Valid id's: 0..9
```

The first 4 lines are comment lines.

The remaining lines define the message protocol in detail by specifying the different message elements. Element descriptions can optionally contain a comment (after the ABNF element description) which can be shown as help text by SyncFolder. Provide a comment by preceding the comment with a semicolon «;». A comment can span multiple lines by starting with a semicolon on each line.

More information about the use of ABNF can be found, amongst others, on Wikipedia and in the official IETF specification RFC 5234.

Examples of ABNF template files can be downloaded from the TestExpert website.

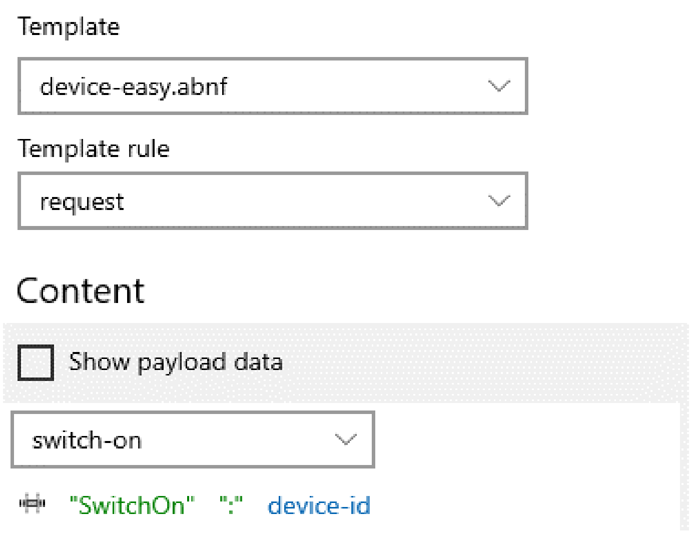
To use the message editor you must first go through the following 2 steps:

- Create an ABNF file containing all the required message rules. The section about Template files later in this document describes the format of the file in more detail and some TestExpert specific extensions.
- Then make sure that the template file has been configured in the solution. You can do this by opening the Solution properties page and adding the ABNF file there.

You can then make use of the template file for both outgoing and incoming messages.

Specifying outgoing messages

When the above template file is used to define for instance a outgoing *SwitchOn* message, TestExpert will present the following interface to the user in its *Template message editor*:


Template

device-easy.abnf

Template rule

request

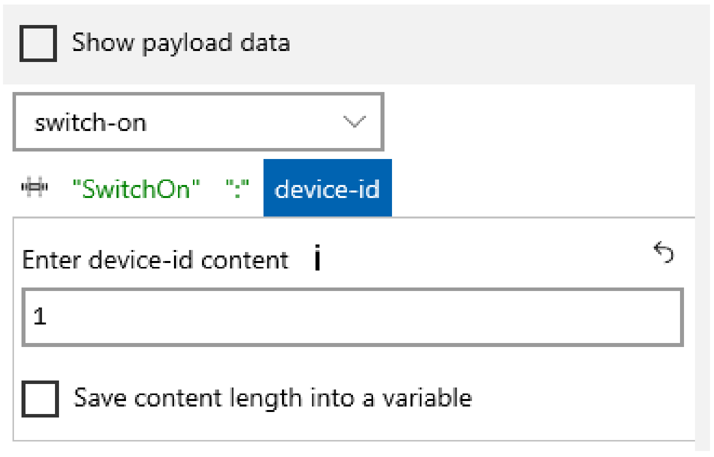
Content

Show payload data

switch-on

"SwitchOn" ":" device-id

Elements shown in green don't need any further specification, elements shown in blue (like device-id in the example) require additional actions. Click on the element to provide the necessary data.


Show payload data

switch-on

"SwitchOn" ":" device-id

Enter device-id content

1

Save content length into a variable

Specification modes

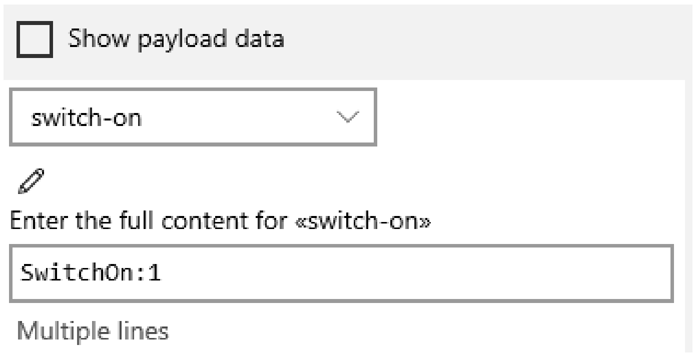
There are 2 ways to define the content of the various message elements that are presented by the ABNF editor when editing a outgoing message:

	Template Mode - Define the content of an element from the ABNF template.
	Edit Mode - Specify the content of the complete element manually in a text box.

The option of your choice can be selected by clicking on the small button on the left of the element header.

1. With the first *Template Mode* option, when you select an element then the editor will take the ABNF template file definitions and show the ABNF element in the following manner:
   - As a single element for which you must define the content in a text box, or
   - As an optional element that you can include or exclude, or
   - As a group of elements for which each of them have to be expanded in order to specify their final content, or
   - As a list of elements where you can add as many element items as you want, or
   - As a set of alternatives to choose from.

2. With the second *Edit Mode* option you simply define the content of the complete element manually. If we would use this option to define the element content in the above example the definition would be:



Show payload data

switch-on

Enter the full content for «switch-on»

```
SwitchOn:1
```

Multiple lines

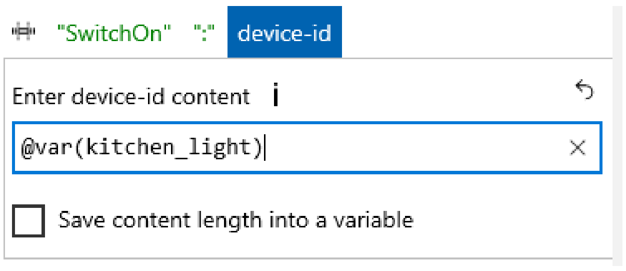
   Because we have chosen not to use the ABNF syntax definition for the “switch-on” element, we must define the content of the whole element ourselves.

Specifying content data in text boxes

There are 2 ways you can specify the content of an element when a text box is presented:

1. Simply define a hardcoded value or string for the content. E.g.: “1”, “Bob”, etc.

2. Enter the value or string by using the TestExpert *@var()* statement to include data taken from static or runtime variables. E.g.:



"SwitchOn" ":" device-id

Enter device-id content

```
@var(kitchen_light)
```

Save content length into a variable

TestExpert will take the specified definition, check if there are TestExpert statements in there and, if so, convert them to their value.
E.g. assume ‘appartment_id’ is a static variable set to “1”, and ‘kitchen_light’ is a variable set to “7”, then the following definition will be translated to “1/7”;

```
@var(appartment_id)/@var(kitchen_light)
```

Single line and multi line text box

The presentation of a text box is depending on whether *Template Mode* or *Edit Mode* is selected.

- In *Template Mode* the default is a single line box, unless the definition of the element in the ABNF template file contains the TestExpert prose-val command <??>, like in:
  `message-body = <??>`

- In *Edit Mode* the default is also a single line box but there is a toggle button under the text box that allows you to switch between a single line and a multi line text box.

A single line text box doesn’t allow you to add newline characters (through the Enter key) whereas a multi line textbox does.

Note: The default character for newline in a multi line textbox is the \n character. You can change that into a \r character followed by \n by opening the Solution Properties page and setting the *“Force line break characters in inputted strings to CRLF..”* toggle switch to ON.

Save content length in a variable

The panel where you can enter data in the text box also shows a *“Save content length into a variable”* checkbox. This can be useful when you don’t want to work with fixed size data but the actual data length must be passed in some other element.

Example: A SIP or HTTP request carries a body message of which the size must be specified in the Content-Length header.

Verify the message definition

Checking the *Show payload data* checkbox will instruct the app to show the complete message data defined by the user.

Note however that if the definition contains TestExpert statements to take data from variables then those statements are not parsed.

To see the completely parsed payload you must use the Message Properties panel on the right column of the page and press the *Verify* button.

Specifying incoming messages

The ABNF message editor for incoming messages looks similar to the one for outgoing messages but presents some additional options that are needed when you want to tell TestExpert to maybe ignore the content of some elements are allow different values for some elements. Without this you must exactly describe all messages upfront what in most cases would not be possible.

The way TestExpert provides this flexibilty is by offering 4 ‘specification modes’ for each of the message elements:

╫	*Template Mode* - Define the expected content of an element from the ABNF template.
✎	*Edit Mode* - Specify the expected content of the complete element manually in a text box.
🔍	*Match Mode* - Match the content using a regular expression. This allows you to specify a partial match, and a match with possible different alternatives.
↦	*Skip Mode* - Skip the content completely and thus accept any content for the field/element.

1. The first 2 options are the same as for an outgoing message.

2. *Match Mode* allows you to validate the incoming message element by providing a regular expression that specifies what data/values are allowed to be present in the content of that particular element.

3. *Skip Mode* allows you to tell the app that you are not interested in the actual data of the content; i.e. any data of the incoming message element is allowed.

When using *Match Mode* or *Skip Mode* the app must be told where it can find the next element when it has finished processing the element. This is done by providing an appropriate regular expression pattern, unless the element is the last element in the message definition.

Note - Because regular expressions are an important tool in TestExpert when dealing with message validation and also information extraction (see later) for ABNF template messages you will need some knowledge about it.
A good information source for this is Wikipedia.
A quick reference of the *Regular Expression Language* (as supported by the .NET engine in TestExpert) can be found here and here.

A good website to verify your regex patterns against content of your choice is .NET Regex Tester - Regex Storm

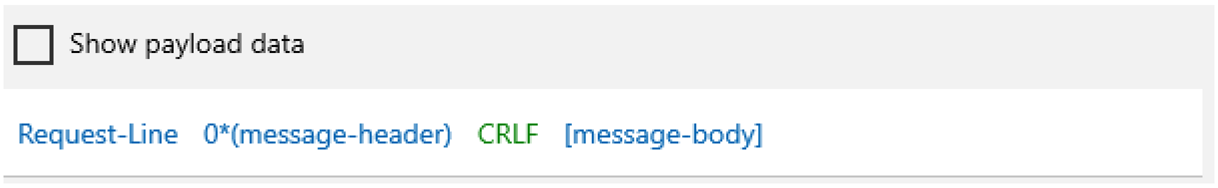
The next examples show how you can use the above 4 specification modes to specify incoming messages completely in all details or partially where you only want to validate a couple of elements and/or want to skip validation of one or more elements.

All examples assume the following SIP input message:

```
INVITE sip:bob@biloxi.example.com SIP/2.0
Via: SIP/2.0/TCP client.atlanta.example.com:5060;branch=z9hG4bK74bf9
Max-Forwards: 70
From: Alice <sip:alice@atlanta.example.com>;tag=9fxced76sl
To: Bob sip:bob@biloxi.example.com
Call-ID: 3848276298220188511@atlanta.example.com
CSeq: 1 INVITE
Contact: sip:alice@client.atlanta.example.com;transport=tcp
Content-Type: application/sdp
Content-Length: 151

v=0
o=alice 2890844526 2890844526 IN IP4 client.atlanta.example.com
s=-
c=IN IP4 192.0.2.101
t=0 0
m=audio 49172 RTP/AVP 0
a=rtpmap:0 PCMU/8000
```

The top ABNF element (retrieved from a SIP template file) that describes such a SIP request message looks as follows:



Request-Line 0*(message-header) CRLF [message-body]

It consists of 4 parts:

- A Request-Line (the 1st line in the above example message) ending with a CRLF.
- A number of message-header lines (the 2nd till 10th line above) each ending with a CRLF.
- A empty line with only a CRLF.
- The message-body consisting of a number of lines.

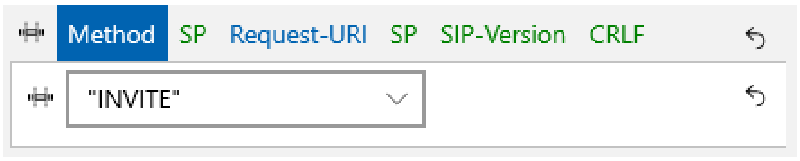
First, lets look at the definition of the Request-Line containing the method (“INVITE”), the request-uri (“sip:bob@biloxi.example.com”) and the SIP-Version (“SIP/2.0”).

In the following sections we show a couple of ways how to define this.

Full Template Mode

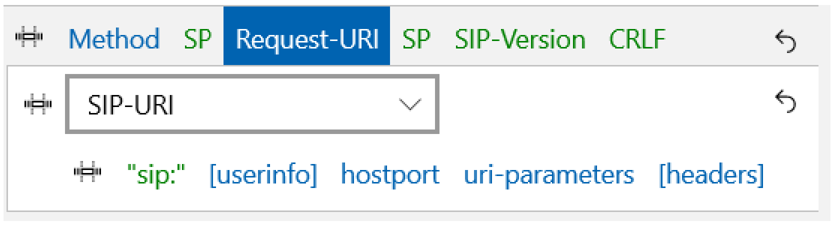
Using *Template Mode* on all the message elements of the expected Request-Line, the definition for the first 2 elements in the Request-Line (the SIP-Version is hardcoded in the ABNF template file) must be:

Method



The method is fixed by the user specifying that an incoming message must start with the “INVITE” string in order to match this message. The string is selected from a dropdown box that holds all possible method strings defined in the ABNF.

Request-URI

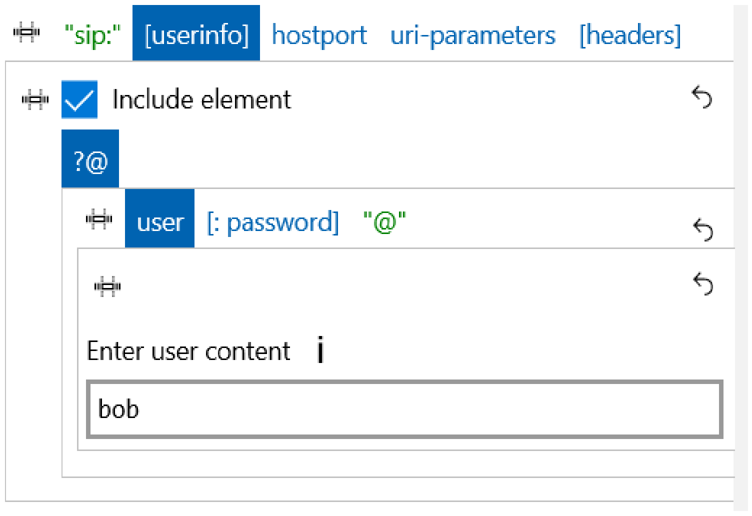

Method SP Request-URI SP SIP-Version CRLF
SIP-URI
"sip:" [userinfo] hostport uri-parameters [headers]

The uri must start with “sip:” and consists of: optional ‘userinfo’, a mandatory ‘hostport’ element, 0 or more ‘uri-parameters’ and optional ‘headers’.

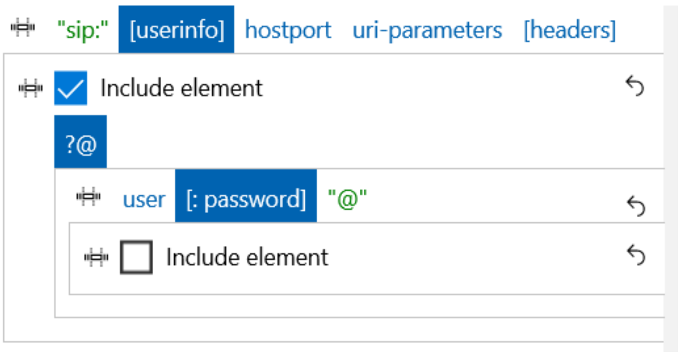
userinfo

In this example we want to validate the userinfo element and tell TestExpert that the user’s name in the incoming message must be “bob” and the optional password must not be present. The userinfo element then must look as shown next.

The ‘user’ part:



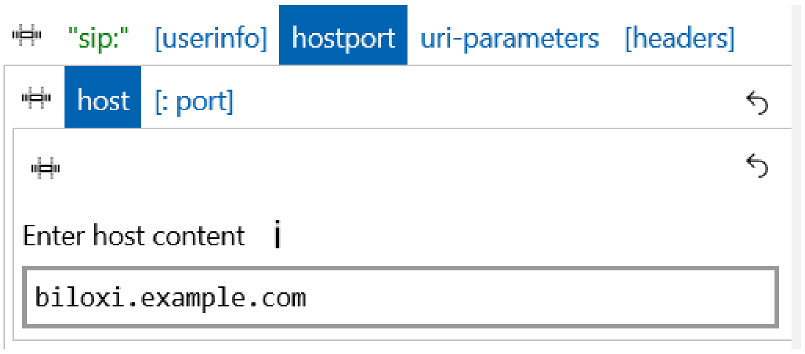
And, the ‘password’ part:



With the above definition a message with a Request-URI that doesn't contain bob in the userinfo part or contains a name and a password, will be indicated as not matching.

Hostport

Hostport consists of a host part and an optional port number.The host is hardcoded by the user as follows:



Enter host content

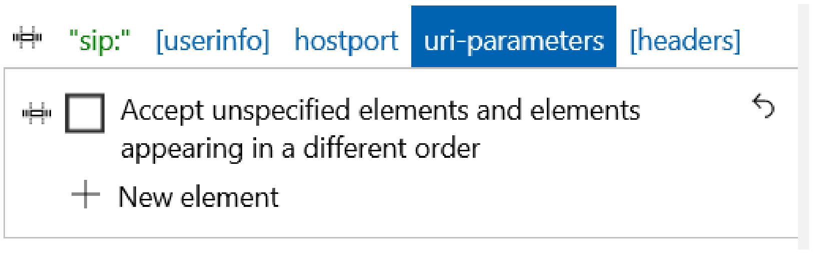
biloxi.example.com

The host port number is excluded. We don't show it here but its 'Include element' checkbox is unchecked.

uri-parameters

Uri parameters in a Request-URI are optional but, when present, you can have more than one. Each parameter starts with a semicolon, followed by a name, an equal sign, and a value.
The list of parameters ends when either a header is reached or the SPACE character (which ends the Request-URI).

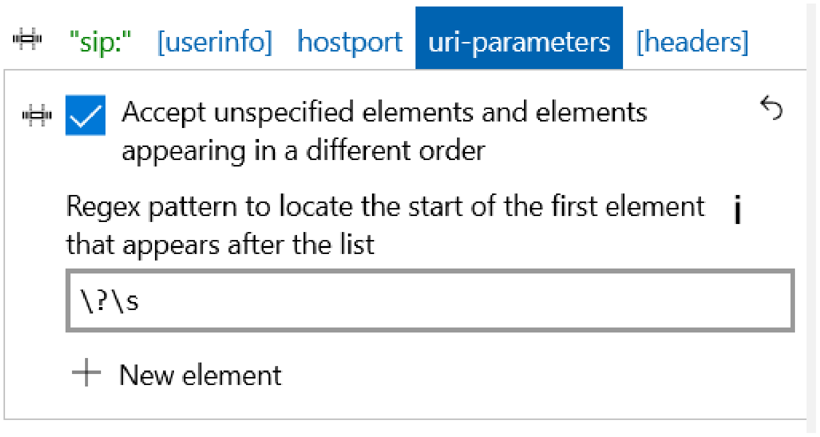
1. If we assume that the Request-URI musn’t contain uri parameters then the uri-parameters element must be defined as follows:


"sip:" [userinfo] hostport uri-parameters [headers]

Accept unspecified elements and elements appearing in a different order

New element

2. If we assume that the Request-URI might contain uri parameters, but we don’t specify any ourselves, then the definition must look as follows:


"sip:" [userinfo] hostport uri-parameters [headers]

✓ Accept unspecified elements and elements appearing in a different order

Regex pattern to locate the start of the first element that appears after the list

\?\s

New element

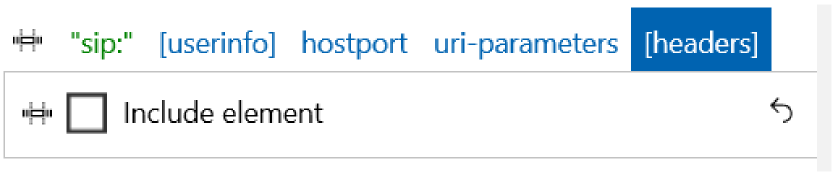
The ‘Accept unspecified elements…’ checkbox must be checked to indicate that we allow unspecified uri-parameters.

The regex pattern must be provided because uri-parameters is a list and so we must tell the app where the list ends. Here the list ends at either a ? character (which is the 1st character of a header when present) or at the SPACE character (which is the end of the complete Request-URI).

The regex pattern therefore is: «\?\s»

headers

We assume that there are no headers in the Request-URI. The definition therefore is:


"sip:" [userinfo] hostport uri-parameters [headers]

Include element

Combination of Template Mode and Skip Mode

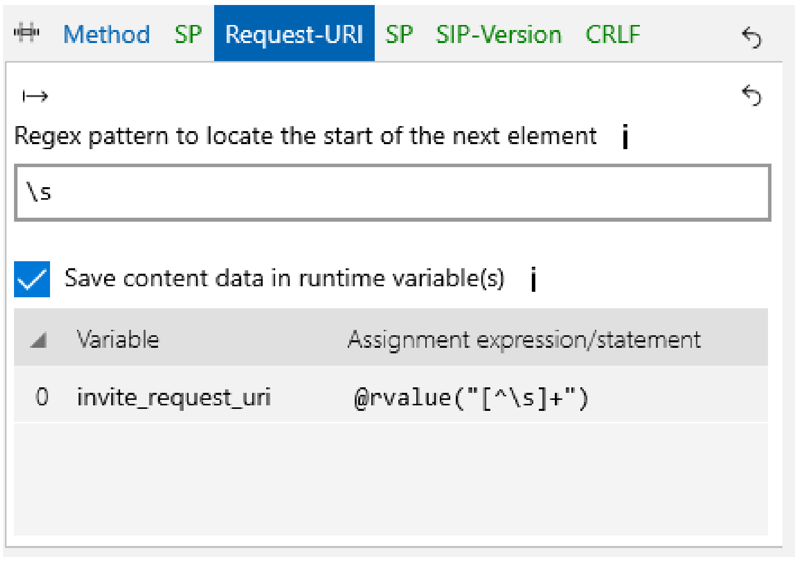
In practise, full template mode would limit the use of the above message in various scenarios since the expected content is almost completely fixed upfront. In most cases that would not be a good idea and so a combination of *Template*, *Match* and

Skip Mode will be better because it offers more flexibility when choosing to either check or ignore certain elements of a message.

The first element that is a candidate for such flexibility would be the Request-Uri.

Request-URI:

By setting *Skip Mode* for the Request-URI, we tell TestExpert to not validate the content of the uri. The definition looks as follows:



Regex pattern to locate the start of the next element

```
\s
```

Save content data in runtime variable(s): ✓

	Variable	Assignment expression/statement
0	invite_request_uri	`@rvalue("[^\s]+")`

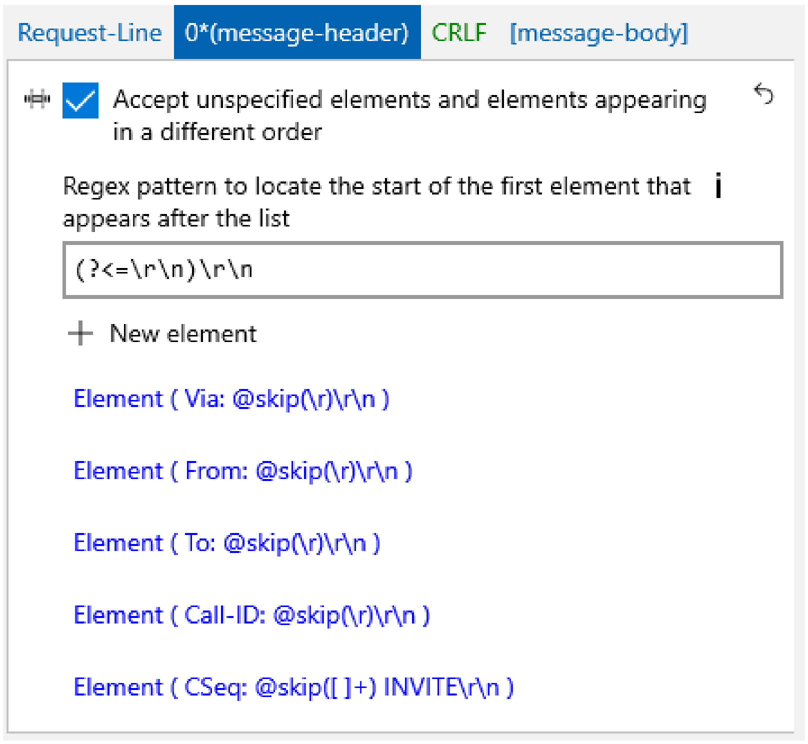
Any element that is told to be skipped and is not the last element in the message needs a regex pattern that tells the app where the next element is located in the incoming message. For a SIP Request-URI, this is the first SPACE character. The regex pattern therefore contains «\s».

Note also the table with the "invite_request-uri" entry. It is there to tell the app to save content of the element in a runtime variable. This can then be used in an outgoing message when sending a response. See ABNF incoming message extraction for more details.

Message headers

The message headers in the incoming SIP message are another example where it makes sense to use both *Template* and *Skip Mode*.

The headers are specified in our ABNF template file by the 0*(message-header) repetition group element and thus are part of an element list.
For our example we want to specify a set of mandatory headers and also indicate that it is allowed to have additional, unspecified headers in the message. The definition for the repetition group / list then looks as follows:



✓ Accept unspecified elements and elements appearing in a different order

Regex pattern to locate the start of the first element that appears after the list

```
(?<=\r\n)\r\n
```

Element (Via: @skip(\r)\r\n)

Element (From: @skip(\r)\r\n)

Element (To: @skip(\r)\r\n)

Element (Call-ID: @skip(\r)\r\n)

Element (CSeq: @skip([ ]+) INVITE\r\n)

The *“Accept unspecified elements…”* checkbox is checked which tells TestExpert to ignore list elements for which the user hasn’t provided a definition and to ignore the order of the element that have been defnied.

Because the group element holds a list of elements we must also tell TestExpert how it can find the end of the list. You do that by entering a regex pattern in the presented text box. In our example we must locate the 1st empty line that follows after the last header and therefore the pattern is set to «(?<=\r\n)\r\n».This pattern indicates that the element that follows the last header is an empty line (with only a CRLF) that follows after the CRLF of the last header.

The next step is to specify which of the SIP headers we definitely want to see in the incoming message. These mandatory headers are: Via, From, To, Call-ID, and CSeq.

Let’s look at the definition of one of those headers.

Message-header

The ABNF definition of message-header is (partially):

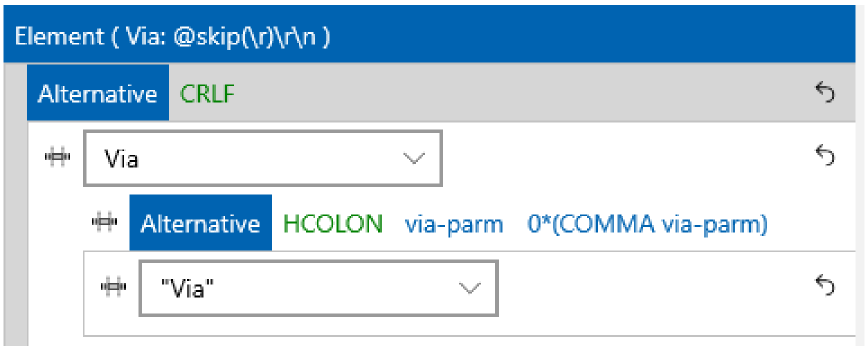
```
message-header = (Accept / Via / To / From / Call-ID
                  / CSeq / Accept-Encoding) CRLF
```

Via header

The ABNF definition of the Via header is:

```
Via = ("Via" / "v") HCOLON via-parm *(COMMA via-parm)
```

The Via header is therefore specified by first selecting the header name from a list of alternatives and then selecting “Via” or “v” as the actual name to be used.


Element (Via: @skip(\r)\r\n)

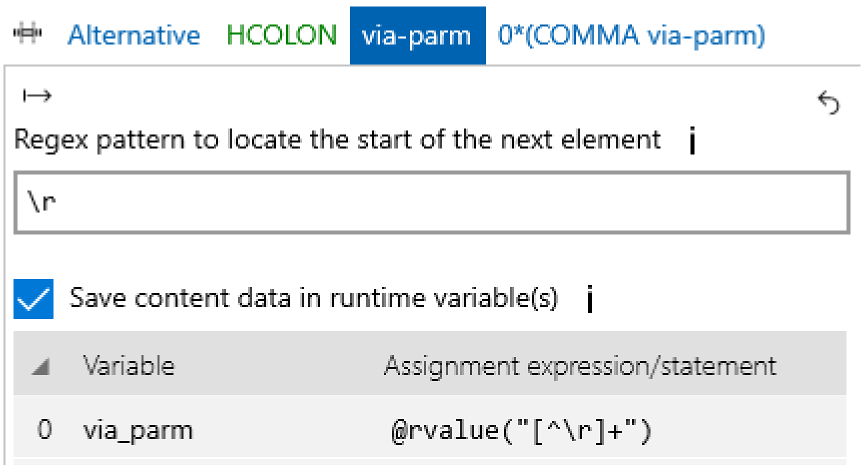
Alternative CRLF

Via

Alternative HCOLON via-parm 0*(COMMA via-parm)

"Via"

The via-parm field that follows the colon character after the “Via” string must not be validated in a TestExpert scenario and therefore its definition is as follows:


Alternative HCOLON via-parm 0*(COMMA via-parm)

Regex pattern to locate the start of the next element

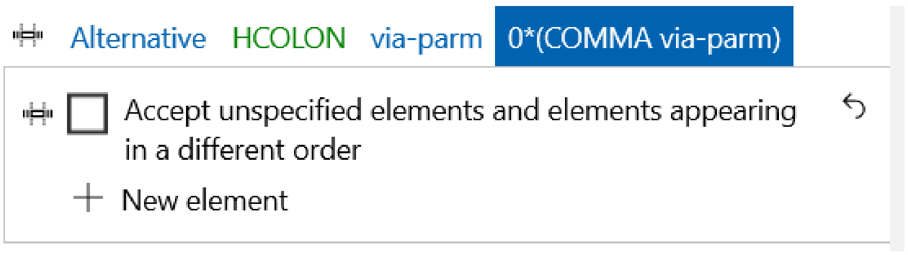
\r

☑ Save content data in runtime variable(s)

	Variable	Assignment expression/statement
0	via_parm	@rvalue("[^\r]+")

The regex pattern for skipping the via-parm element is set to «\r» which also indicates that the next 0*(COMMA via-parm) element in the definition will be unchecked and will not have additional via-parm elements.

The definition for the 0*(COMMA via-parm) element therefore is:


Alternative HCOLON via-parm 0*(COMMA via-parm)

☐ Accept unspecified elements and elements appearing in a different order

New element

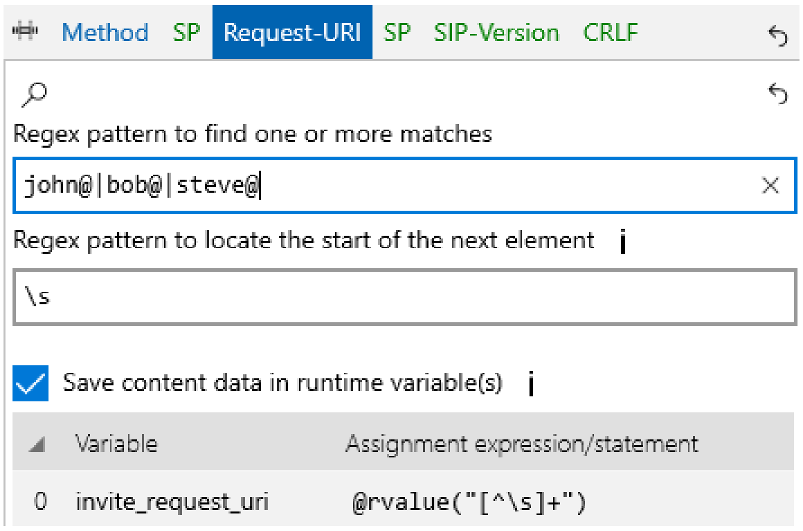
To summarize: The above definitions indicate that a Via header starts with “Via”, then follows a colon, then follows one or more via-parm strings that are not checked, then follows a CRLF.
The table entry indicates that everything that follows after the colon (excluding the terminating CRLF) is saved in runtime variable “via_parm”.

Match Mode

Match Mode must be used when you don’t want to define upfront the full content of an element, or don’t want to simply skip the content, but want to instruct TestExpert to validate some data in the incoming message element.

An example: In the previous section the content of the SIP Request-URI was not validated. Now we want to check if the username field is set to “john”, “bob”, or “steve”.

The simplest way to do that is to switch on *Match Mode* for the complete Request-URI and define the element as follows:



Regex pattern to find one or more matches

```
john@|bob@|steve@
```

Regex pattern to locate the start of the next element

```
\s
```

✓ Save content data in runtime variable(s)

	Variable	Assignment expression/statement
0	invite_request_uri	`@rvalue("[^\s]+")`

There are 2 text boxes here:

The 1st text box must get a regex pattern that describes which input data is considered to be valid.

The 2nd text box is the same as with *Skip Mode* and therefore must contain a regex pattern to locate the end of the element.

Often used regexps for message validation

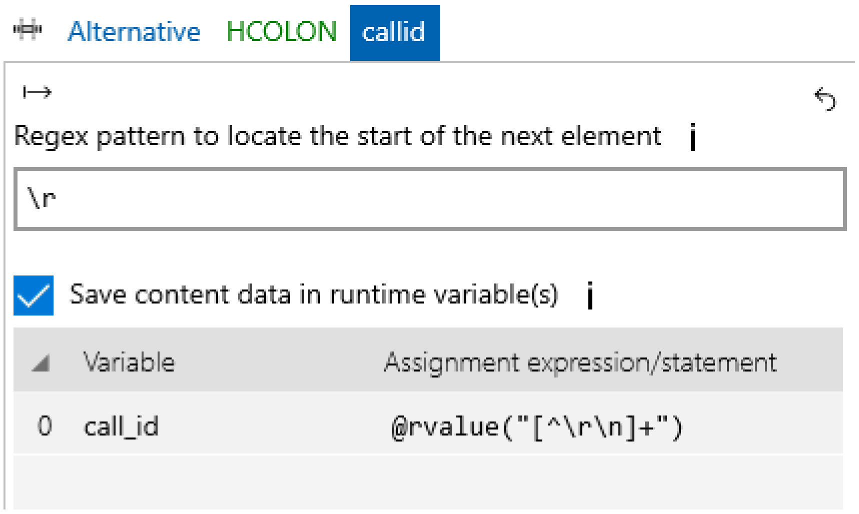
`.*`	Skip till the next newline/LF (\n, 0x10) character (without consuming the LF).
`.*(?=\r)`	Skip till the next CR (\r, 0x0A) character (without consuming the CR). Useful when a newline is specified by means of a CRLF.
`.+(?=\r)`	Expect at least 1 character and skip then till the next CR (\r, 0x0A) character (without consuming the CR). Useful when a newline is specified by means of a CRLF.
`.+(?= )`	Expect at least 1 character and skip then till the next space/SP (0x20) character (without consuming the SP character).
`\r\n(?=\r\n)`	This searches the content for 2 consecutive CRLF's but without 'consuming' the 2nd CRLF. It makes use of a regular expression look-

	behind assertion and skips/delimits everything from the current position till the position in the string where there is a CRLF (0x0A, 0x10) immediately following another CRLF. The second CRLF will not be part of the delimited area.
`(.|\s)*`	Skip all remaining characters of the string (including whitespace: space, tab, newline, etc.).
`(\s*)`	Skip all whitespace characters (if any).

Save data from incoming messages

The ABNF message editor allows you to tell the app to extract incoming message data while a scenario executes and save this data in runtime variables. The saved data can then be used to format a possible reply message.

To use this feature for a given message element you must configure *Match Mode* or *Skip Mode* on that element. With those 2 modes you get a checkbox that you must check in order to see a table where you can add statements for extracting and saving data in variables, like so:



Regex pattern to locate the start of the next element: `\r`

☑ Save content data in runtime variable(s)

	Variable	Assignment expression/statement
0	call_id	`@rvalue("[^\r\n]+")`

There are 3 columns:

- The 1st column contains the row number. The number is not important but the cell is there to allow you to add/delete rows. Do a right click on the row number to see a flyout menu with Cut, Copy, and Insert commands.
- The 2nd column / cell contains the name of the runtime variable that has to be created/updated with the value of the retrieved data.
- The 3rd column/cell will typically contain the TestExpert @rvalue(), @xvalue() or @jvalue() assignment statement to indicate which data must be retreived from the incoming message element.

The 3 assignment statements:

1. When the incoming message contains a unformatted string then you must use the **@rvalue(pattern)** statement. This tells TestExpert to use the pattern in a

regular expression in order to match the required data and save the complete match in a runtime variable.

2. When the incoming message contains content that is formatted in XML then you can use **XPATH** to select nodes and store the associated data in a runtime variable.
   You will use the MDL statement **@xvalue(path)** for this.
3. When the message contains JSON content then you can use **JSONPath** to select nodes and store the associated data in the runtime variable.
   You will use the MDL statement **@jvalue()** for this.

Message extraction using @rvalue() with a regex

The extraction pattern must be given as a 'string' parameter between double quotes in the @rvalue() statement and is handled by TestExpert during message parsing as follows:

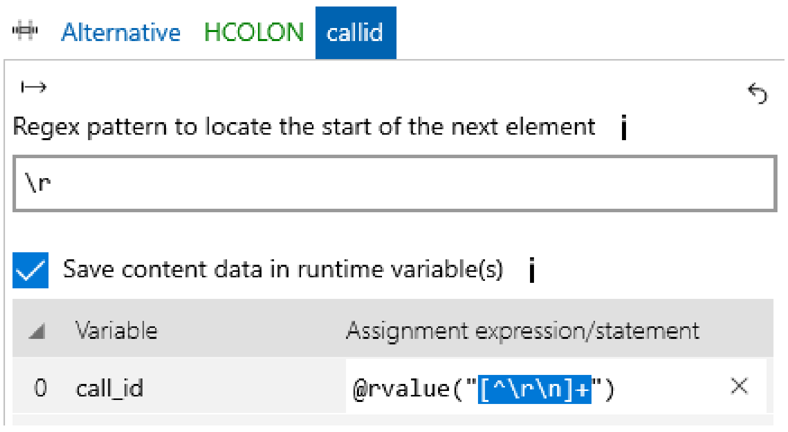
- TestExpert executes a regex with the assigned pattern with the starting search position set to the 1st content character of the incoming element.
- When it finds a match it copies, starting from the match index, as many characters to the variable as indicated by the length of the match.

Example 1

A first example shows how to extract the complete value of the Call-ID header from our incoming SIP message. This header looks as follows:

```
Call-ID: 3848276298220188511@atlanta.example.com
```

The call-id value is everything after the semicolon «:». To extract this value the table entry for a variable 'call_id' would be like this:



Regex pattern to locate the start of the next element

```
\r
```

✓ Save content data in runtime variable(s)

	Variable	Assignment expression/statement
0	call_id	`@rvalue("[^\r\n]+")`

The regex pattern is set to «[^\r\n]+». This means: take all characters but stop when a \r or \n character is detected.

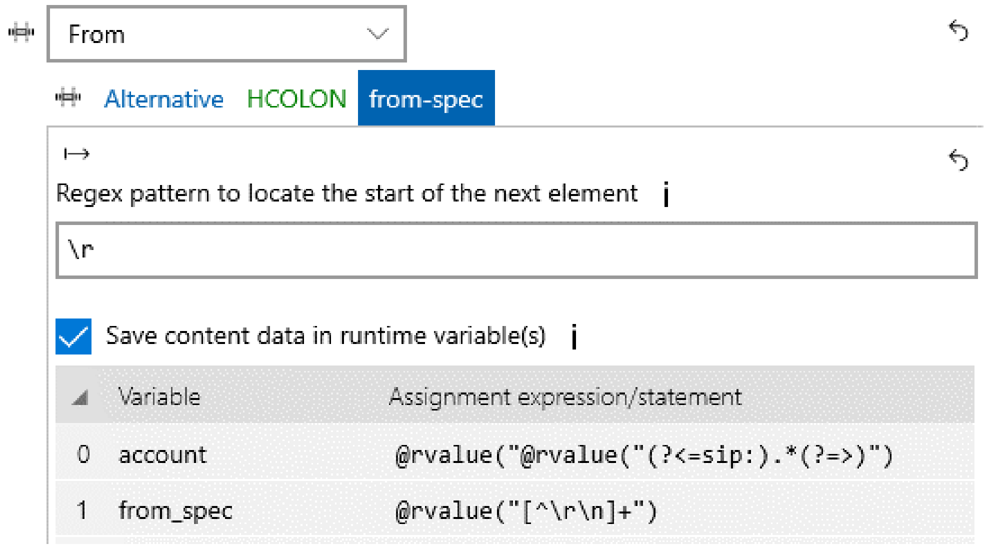
This will result in the variable 'call_id' getting the value "`3848276298220188511@atlanta.example.com`".

Example 2

Assume you want to save the account address that appears in the 'From' header of our example message in a runtime variable with the name 'userAccount'. This header looks as follows:

```
From: Alice <sip:alice@atlanta.example.com>;tag=9fxced76sl
```

The accountname is everything between «<sip:» and «>». To extract this name the @rvalue() pattern for ‘account’ must looks as follows:



Regex pattern to locate the start of the next element

```
\r
```

✓ Save content data in runtime variable(s)

	Variable	Assignment expression/statement
0	account	`@rvalue("@rvalue("(?<=sip:).*(?=>)")`
1	from_spec	`@rvalue("[^\r\n]+")`

Message extraction using @jvalue() with JSONPath

Assume the following JSON result in a message:

```
{ "store": {
      "book": [
        { "category": "reference",
            "author": "Nigel Rees",
            "title": "Sayings of the Century",
            "price": 8.95
        },
        { "category": "fiction",
            "author": "Evelyn Waugh",
            "title": "Sword of Honour",
            "price": 12.99
        },
        { "category": "fiction",
            "author": "Herman Melville",
            "title": "Moby Dick",
            "isbn": "0-553-21311-3",
            "price": 8.99
        },
        { "category": "fiction",
            "author": "J. R. R. Tolkien",
            "title": "The Lord of the Rings",
          "isbn": "0-395-19395-8",
          "price": 22.99
        }
      ],
      "bicycle": {
        "color": "red",
        "price": 19.95
      }
  }
}
```

Some examples

JSONPath	**Result**
`$.store.book[*].author`	The authors of all books in the store.
`$..author`	All authors.
`$.store.*`	All things in store, which are some books and a red bicycle.
`$.store..price`	The price of everything in the store.
`$..book[2]`	The third book.
`$..book[-1:]`	The last book in order.
`$..book[0,1]`	The first 2 books.
`$..book[:2]`	The first 2 books.
`$..book[?(@.isbn)]`	Filter all books with isbn number.
`$..book[?(@.price<10)]`	Filter all books cheaper than 10.
`$..*`	All members of the JSON structure.

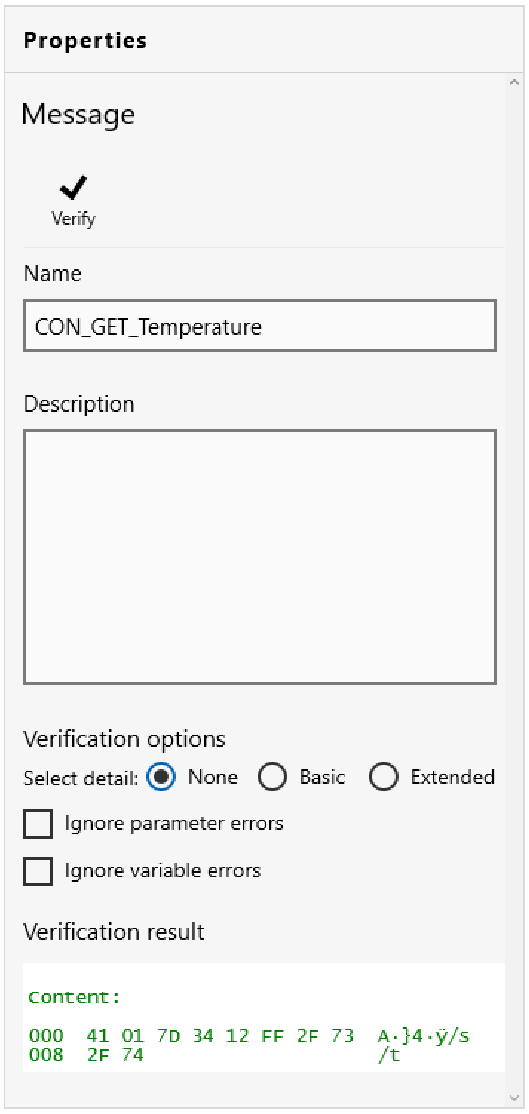
Message extraction using @xvalue() with XPath

Still to do.

Message properties editor

The *Message properties editor* allows you to configure a number of properties of the message and have a look at how the message data would look like when it would be sent out. The editor comes up in a number of ways:

- When you have a message open in TestExpert's tabs area and the screen is wide enough the *Message properties editor* opens automatically on the right side of the screen.
- When the screen is too narrow you can open the editor by pressing the *Properties* button in the Message editor view. In this case the editor shows in front of the tabs area. You close it by pressing the *Back* arrow icon in the top left corner of the windows.



Properties

Message

Name

CON_GET_Temperature

Description

Verification options

Select detail: [✓ None] Basic Extended

☐ Ignore parameter errors

☐ Ignore variable errors

Verification result

```
Content:
000  41 01 7D 34 12 FF 2F 73  A·}4·ÿ/s
008  2F 74                    /t
```

Name

You can change the name of the message by tapping the *Name* text box and then typing/overwriting the name.

Description

The annotation box is a simple multi-line edit control in which you can type any description you want. Its usage is optional.

Verify

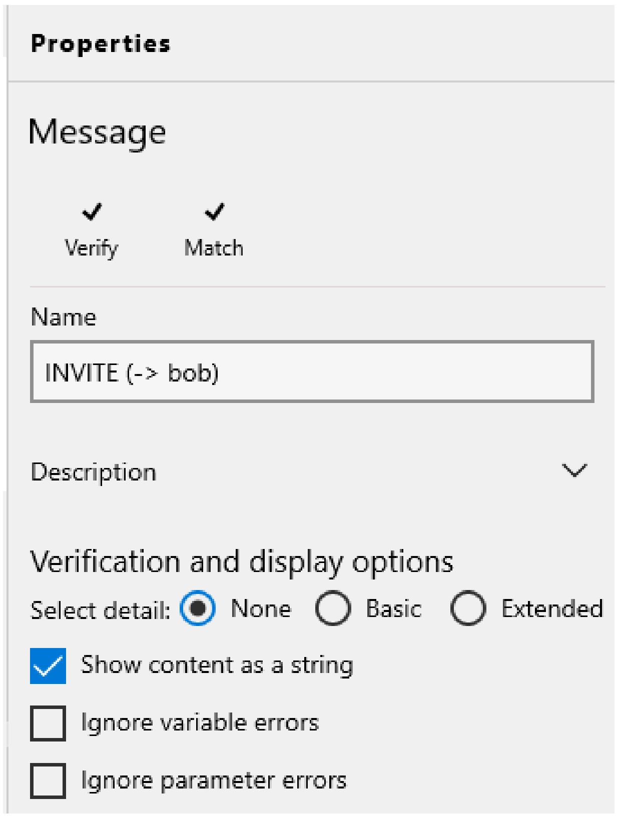
You can ask TestExpert to parse the message definition as it is currently set in the *Message editor* into a list of values by tapping or clicking the *Verify* button. When done the message is shown standard as a set of hexadecimal bytes . If you like to see the resulting data as a string then you can opt for that by checking the '*Show content as string'* checkbox.

When everything is well the message values are shown in green.

When TestExpert is not able to decode a given definition (e.g. because the user hasn't provided a concrete specification for it or the definition contains errors) the message output stops at the faulty location and text and values are shown in red.

Match

When looking at the properties of an incoming ABNF type of message there is also a *Match* button which you can click to match the message definition against message content that you can provide in a dialog window that will pop up.


Properties

Message

✓ Verify ✓ Match

Name

INVITE (-> bob)

Description

Verification and display options

Select detail: [✓ None] Basic Extended

☑ Show content as a string

☐ Ignore variable errors

☐ Ignore parameter errors

Enter the message content or paste the content from somewhere in the text box of the dialog window and press the *Apply* button.

TestExpert will then try to match the provided content against the message definition. When the message definition contains one or more entries to create runtime variables from the inputted content, and the match is OK, then TestExpert will create/update these variables. You can then use those variables to verify outgoing messages that make use of them.

Verification and Match options

Select *None*, *Basic* or *Extended* in the *Select detail* radio buttons group to show less or more information when requesting *Verify* or *Match*.

Check the *Ignore parameters error* and *Ignore variable errors* checkboxes to prevent a faulty dump in case you would have definitions in your message which refer to parameters and/or variables that might not be defined at this stage.

Managing raw message elements

Note: This chapter is only applicable when dealing with raw messages, NOT with HTTP or ABNF type of messages!

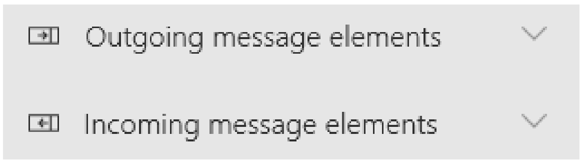
Messages that need to be exchanged between 2 devices are formatted following a well-known protocol specification that defines the structure and the payload of the messages. The structure is typically defined in terms of message fields which contain different classes of information. These fields can present information about the message originator, the message recipient, a subject matter, possibly references to previous messages, user data, etc.
Sometimes these fields can be as simple as a string or a byte but often they are more complex, e.g. in binary message protocols.

In order to avoid that you have to re-specify such fields completely again in different messages the TestExpert solution file comes with a separate *Outgoing message elements* and *Incoming message elements* section where you can define individual information elements (with a name) and specify their content. TestExpert then allows you to ‘include’ such a reusable element at the place you want when you are defining an outgoing or an incoming message.

You can create 2 type of message elements:

- Elements whose content is fully specified upfront in the element content definition.
- Elements whose content is depending on one or more input parameters that have to be defined at the moment the element is to be inserted.

Message elements are added to the solution in the same way as real, full messages; i.e. by tapping or clicking on the navigation panel.



Message element editor

Because this feature is particularly important when you are editing a message in ‘raw’ format and of little use when you are using ‘templates’, the *Message element editor* comes with the same user interface as the Raw message editor.

Including Message elements in a Message

The possibility to include a message element is only available in the Raw message editor.

Parameter-less elements

Assume we have a message element called ‘bearerCap’ that is defined in folder ‘edss1’ of the *Outgoing message elements* group. Its content is defined as follows:

Field	**Definition**
IE ident	`@var(eBEARER_CAP)`
length	`0x03`
Octet 3	`0x80`
Octet 4	`0x90`
Octet 5	`0xA3`

You can include the above element as follows:

Field	**Definition**
…	`...`
import bearerCap	`@element("ome/edss1/bearerCap")`
…	`...`

Parameterized elements

Assume we have a parameterized message element called ‘callRef’ whose content is defined as follows:

Field	**Definition**
Call ref length	`0x02`

Call ref	`@if(@this.parm(1))`
	`? (@this.parm(2) + 0x8000)@word`
	`: (@this.parm(2))@word`

The element expects 2 input parameters:

- Parameter 1 is a flag (which can be 0 or 1).
- Parameter 2 is a numeric value (max. 15 bits).

You can include the above element in 2 ways:

Method 1 – Pass parameters within the @element() statement

Field	**Definition**
Protocol discr	`0x08`
import callRef	`@element("ome/edss1/callRef", 0, 7)`

Message type	`@var("eCONNECT")`
…	`...`

Method 2 – Use @setparm()

Field	**Definition**
Protocol discr	`0x08`
set 1st argument	`@setparm(1, 0)`
set 2nd argument	`@setparm(2, 7)`
import callRef	`@element("ome/edss1/callRef")`
Message type	`@var(eCONNECT)`
…	`...`

TestExpert variables

TestExpert provides 2 type of variables which you can use when you want to define the content of incoming messages and outgoing messages, or want to setup a condition for a scenario action:

- Static variables, and
- Runtime variables

Static variables

The purpose of static variables is to define a single value or a set of values, give this value (or values) an identifier name and use that identifier as a replacement for the value(s).
The variables can be referenced when defining the contents of incoming messages and outgoing messages by using the @var statement.

Static variables are also saved in the solution file.

Example 1

Assume you have a static variable called 'AckID' which contains a 16 bit value `0x8001`. It is used in a message as follows (the example is for a 'raw' message definition but variables are also supported with HTTP messages and ABNF template messages):

Field description	**Definition**
ID	`@var(AckID)`
Device #	`0x77`

The resulting message content will then be the following 3 consecutive bytes (assuming *big endian* for word, integer and long values):

```
0x80 0x01 0x77
```

Example 2

The following example shows some other ways to assign values to a variable:

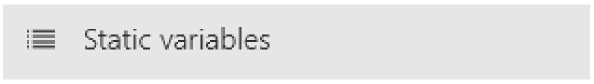
Field description	**Definition**
SIPBranchParm	`"z9hG4bK" @randomstr(11)`
DeviceBase	`0x10`
KitchenLight	`(@var(DeviceBase) + 1)`

The resulting value for SIPBranchParm will be: `"z9hG4bKzckiesJdXR8"`

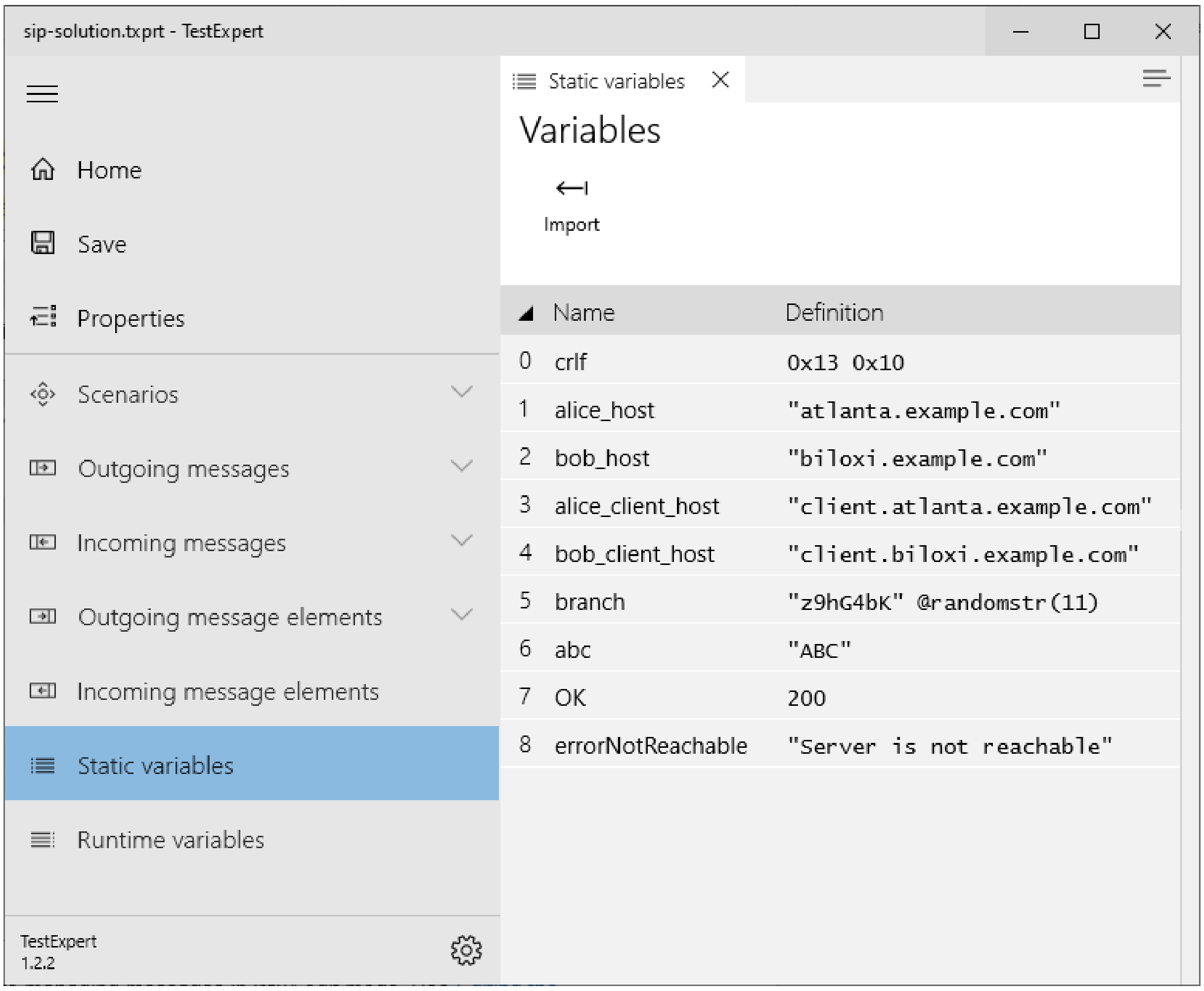
The resulting value for KitchenLight will be: `0x11`

Static variables editor

The *Static variables editor* is opened in a tab item when you tap or click *Static variables* in TestExpert's navigation panel.



Every variable is listed in a row of a table; the name of the variable is in the 2nd column, the value of the variable is defined in the 3rd column.



	Name	Definition
0	crlf	0x13 0x10
1	alice_host	"atlanta.example.com"
2	bob_host	"biloxi.example.com"
3	alice_client_host	"client.atlanta.example.com"
4	bob_client_host	"client.biloxi.example.com"
5	branch	"z9hG4bK" @randomstr(11)
6	abc	"ABC"
7	OK	200
8	errorNotReachable	"Server is not reachable"

Adding, modifying and deleting variables is done in the same way as with message fields when you are managing messages in 'raw' edit mode. See Editing the message table for a description on how to add/delete rows, navigate through the table, enter text in table cells, and do cut/copy/paste actions.

Variables can also be added by clicking the 'Import' button. You will prompted to select a file that contains one or more variable defnitions. The format of this text file is as follows:

```
! A comment
var_1 = def_1
...
var_n = def_n
```

The file type must be .properties or .txt.

Example file

```
! General
abc = "ABC"
OK = 200
! Error strings
errorNotReachable = "Server is not reachable"
```

When importing the app will overwrite existing variables that have the same name.

Runtime variables

Runtime variables are variables which are not stored in the solution file but which can be created by the system when a scenario runs. You can tell TestExpert to create these run-time variables in a number of ways:

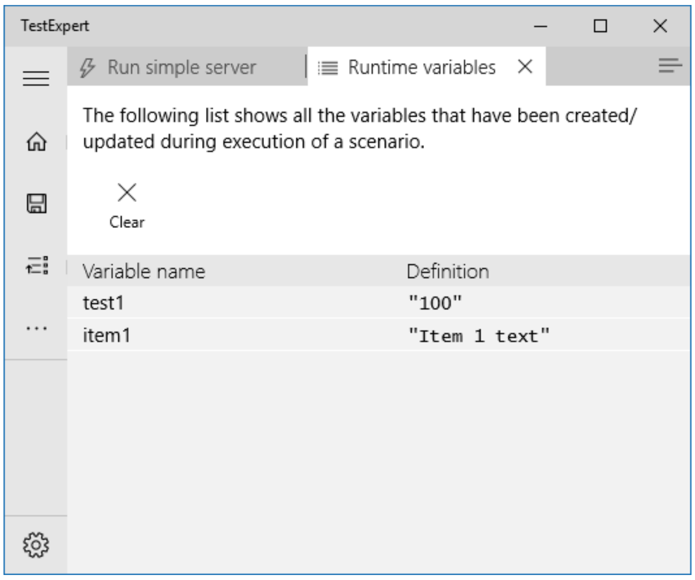
- By defining a @set statement in a *Raw* incoming message that is assigned to a scenario event. When a message comes in that matches the event then TestExpert will create/overwrite the runtime variable and assign the value to it that is specified in the *@set* statement.
- By specifying for an element of a *HTTP* or *ABNF* type of incoming message that one or more variables have to assigned.
- By having an Assign action in a scenario. This sets the runtime variable at the moment the action fires.

You can then refer to these variables in your incoming and outgoing message definitions in the same way as you do with static variables; i.e. by using the @var statement.

Runtime variables view

You can see which runtime variables have been created by tapping or clicking the *Runtime variables* menu item in the navigation panel.

This opens a tab item in the tabs area that shows all runtime variables in a similar table as the static variables table.



The following list shows all the variables that have been created/ updated during execution of a scenario.

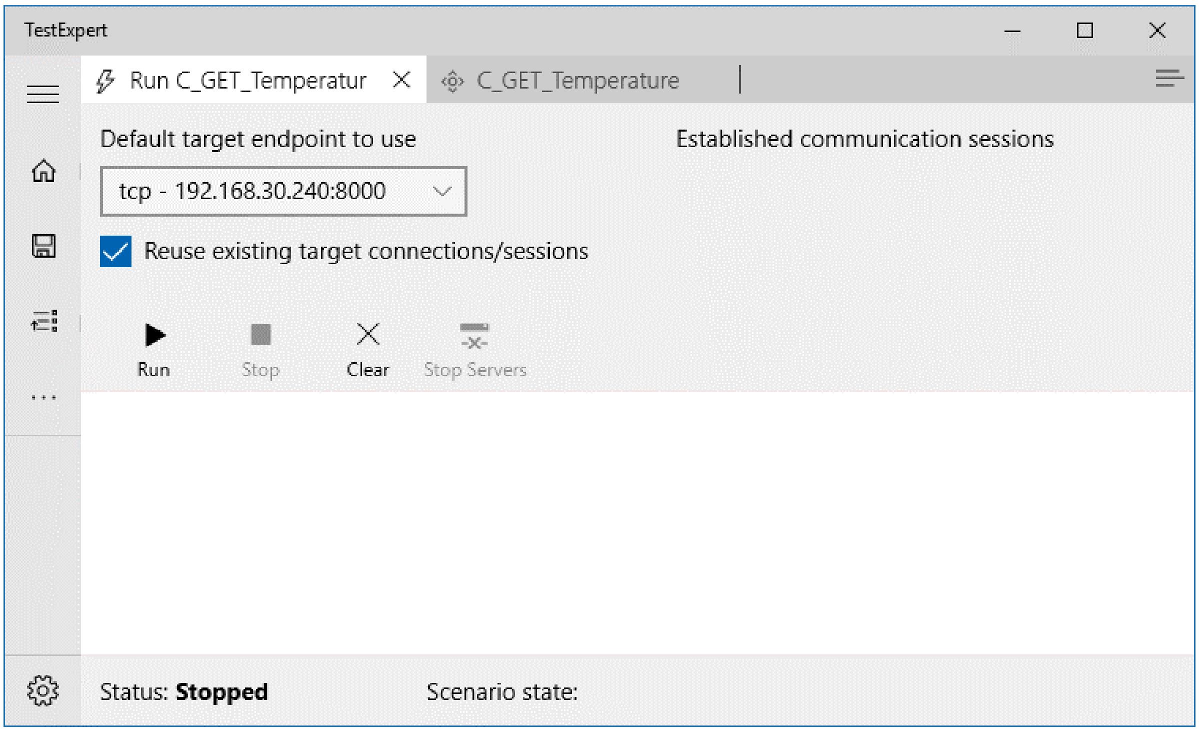
Variable name	Definition
test1	"100"
item1	"Item 1 text"

You can’t change any of the runtime variables but you can clear the whole table.

Running a scenario

To run a scenario you must first open the scenario in the main tab area. Tap or click then the *Run* button in the scenario.

This opens a new tab item that will have as its name 'Run ' followed by the scenario name.



Default target endpoint to use

tcp - 192.168.30.240:8000

✓ Reuse existing target connections/sessions

Established communication sessions

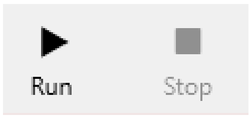
Status: **Stopped**

Scenario state:

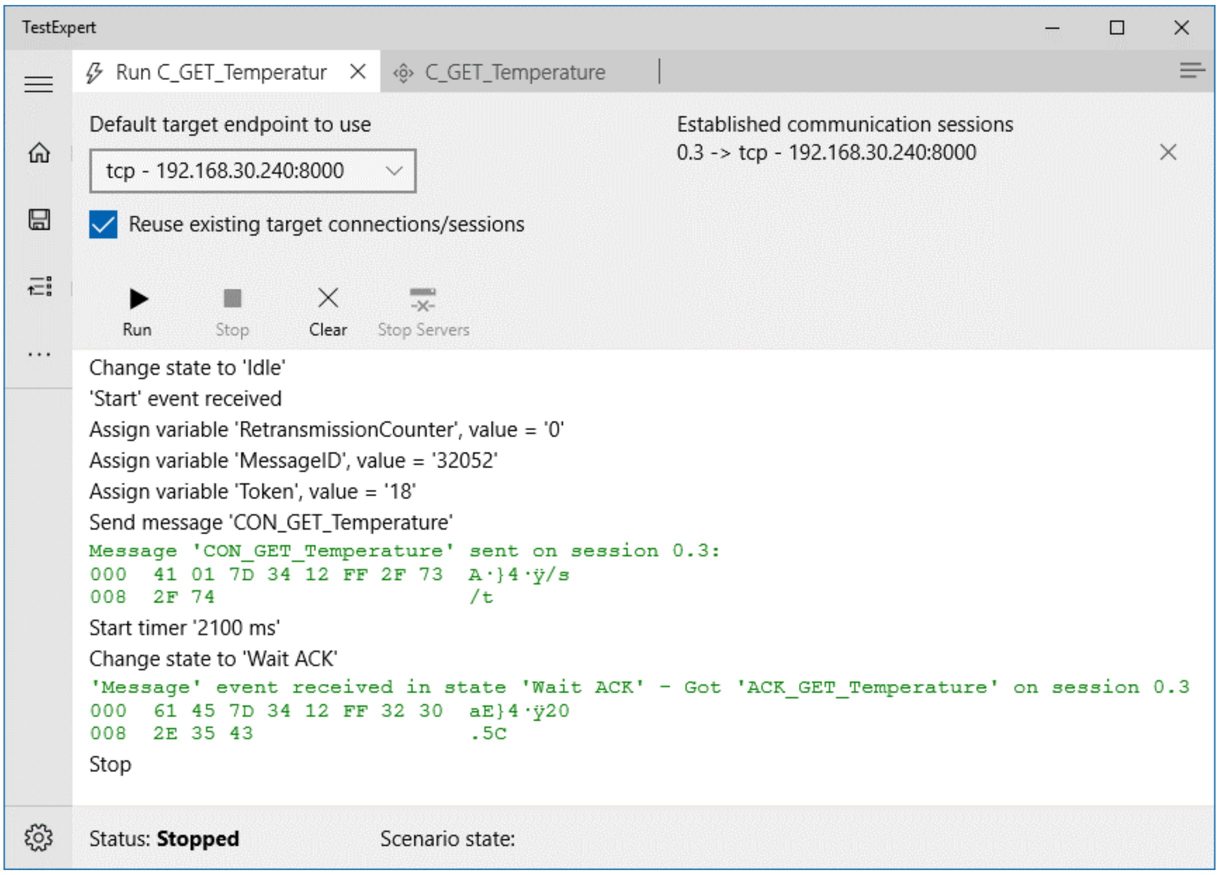
The dropdown box *Default target endpoint to use* lets you select which endpoint you want to use when a message must be send for which no target endpoint has been defined and the message is not a response or a reaction to an incoming message. Target endpoints can be configured in the Solution properties page.

You can check or uncheck *Reuse existing target endpoint connections* to instruct TestExpert to use an already existing connection when sending a message to a target endpoint or create a new connection the first time a message must be sent.

Execution of the scenario can be controlled by the 2 buttons: *Run/Pause* and *Stop*.



After pressing the *Run* button execution of the scenario is started. The history window will then be populated with different kind of messages to show the progress; the events occurring, the actions being executed, etc. The payload of all outgoing and incoming messages will be dumped in a hexadecimal format with valid messages shown in green and invalid messages in red.



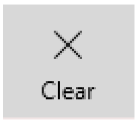
Default target endpoint to use
tcp - 192.168.30.240:8000
Reuse existing target connections/sessions

Established communication sessions
0.3 -> tcp - 192.168.30.240:8000

```
Change state to 'Idle'
'Start' event received
Assign variable 'RetransmissionCounter', value = '0'
Assign variable 'MessageID', value = '32052'
Assign variable 'Token', value = '18'
Send message 'CON_GET_Temperature'
Message 'CON_GET_Temperature' sent on session 0.3:
000  41 01 7D 34 12 FF 2F 73  A·}4·ÿ/s
008  2F 74                    /t
Start timer '2100 ms'
Change state to 'Wait ACK'
'Message' event received in state 'Wait ACK' - Got 'ACK_GET_Temperature' on session 0.3
000  61 45 7D 34 12 FF 32 30  aE}4·ÿ20
008  2E 35 43                 .5C
Stop
```

Status: **Stopped**   Scenario state:

You can clear the history window by pressing the *Clear* button.



Communication using default endpoint

When an outgoing message has been configured to use the default endpoint then the following principles apply:

- If you have a scenario that starts by sending out a message then you must select the endpoint that has to be used for sending from the ‘Default target endpoint to use’ dropdown box.
- If you have a scenario that starts by waiting for a message to come in then you must create a ‘Start server’ action for the “Start’ event and configure a listening endpoint there.
- Once the initiating scenario has sent a message or the listening scenario has received a message, all subsequent messages that are configured to use the default endpoint will be sent to the initially established endpoint or, in case of the listening scenario, to the endpoint that has been created when the first message was received.

Incoming message validation

When a message is received while a scenario is running, TestExpert takes all the message data above the transport layer and tries to find a match with any of the incoming messages that are attached to the current scenario state. That is easy to accomplish when you can define all of your incoming messages as having a fixed, pre-determined, hard-coded set of values.

In most cases however it will probably be impractical to fully specify the expected content of incoming messages. The reason for that is clear:

- Messages will have unpredictable values.
- The sequence of the various information elements is often unspecified.
- Messages will carry less or more optional elements, etc.

To cope with that TestExpert allows you to attach special validation statements and properties to incoming messages. How this has to be done and how the system then tries to find matches is depending on how the content of an attached message is defined: in raw format, as a HTTP message, or as an ABNF formatted message.

Full details on how to deal with this can be found in the description of the various message editors.

Extracting data from incoming messages

When a message comes in on a scenario and matches with an event that is attached to the current scenario state TestExpert will not only execute the attached actions but will also check if the attached message contains statements or properties to update the values of runtime variables. The new values can be:

- A fixed, hard-coded value such as: rtVar1 = “Got it”.
- A dynamic/’calculated’ value such as: rtVar2 = (@var(count) + 1)
- A value extracted from the received message using a regular expression, a XPATH expression or a JSON expression, such as: rtVar3 = @rvalue(“.+(?=\r”)

The method how to specify such statements and how data is extracted is depending on whether we deal with a raw incoming message definition, a HTTP message, or a template based definition. Full details on how to deal with this can be found in the description of the various message editors.

Running a scenario using the Windows Command Line

You can use the windows command line to invoke TestExpert and instruct the app to immediately execute a scenario. You can use this from the command prompt or from the windows task schedular.

The syntax is:

```
testexpert --run [scenarioFolder\]scenarioName
 --solution path\solution.txprt
```

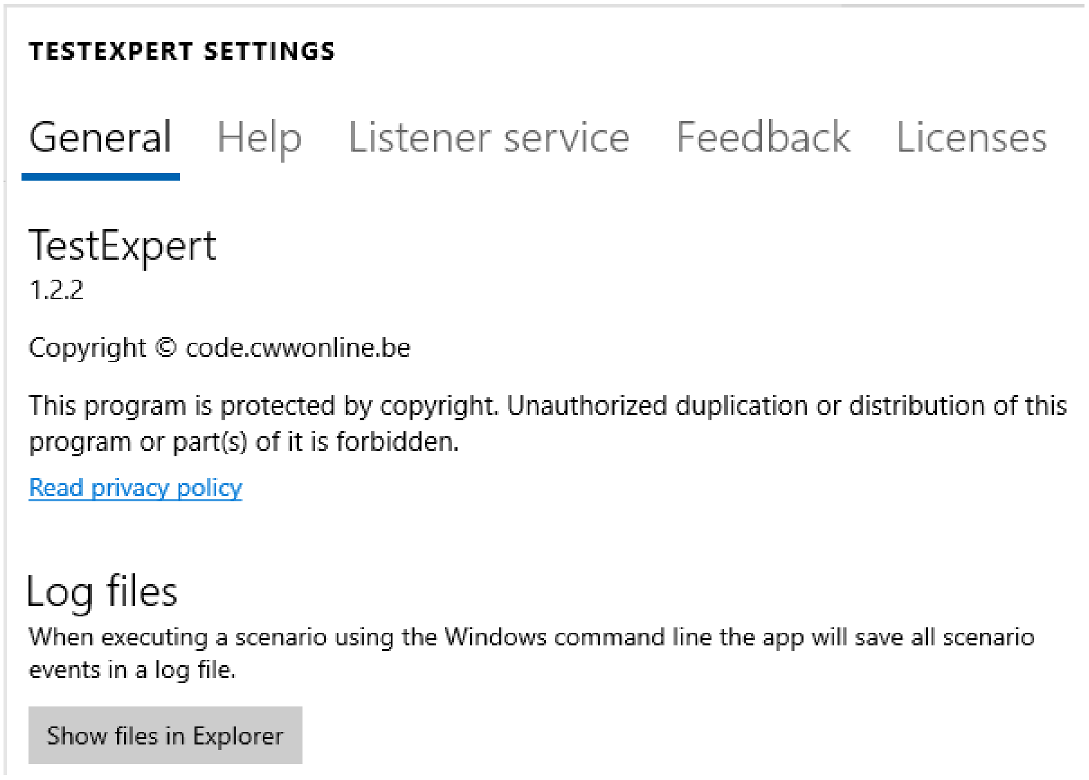
The solution file must be specified using an absolute path notation. Relative or implicit path definitions are not possible.

Examples:

```
testexpert --run "sip client\scenario 3.1 (out)"
 --solution D:\projects\testexpert\sip-solution.txprt

testexpert --run sipclient\scenario_3.1_out
 --solution "C:\Users\me\my projects\sip-solution.txprt"
```

When called from the command line the app will save all scenario events in a log file located in the app’s local folder. You can find the location of this folder by opening the Settings view and clicking the “Show files in Explorer’ button.


TESTEXPERT SETTINGS

TestExpert
1.2.2

Copyright © code.cwwonline.be

This program is protected by copyright. Unauthorized duplication or distribution of this program or part(s) of it is forbidden.

Log files

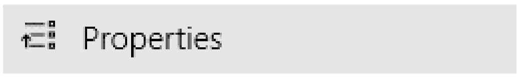
When executing a scenario using the Windows command line the app will save all scenario events in a log file.

Note also that there is no meaningfull exit code for the command line execution. You will have to check the log file or the scenario execution view panel to see if the scenario has executed successfully.

Solution properties page

You can configure a number of settings in the solution file which TestExpert will apply to all included scenarios and message definitions.

Open the *Solution properties page* by tapping the properties menu item in the navigation panel.

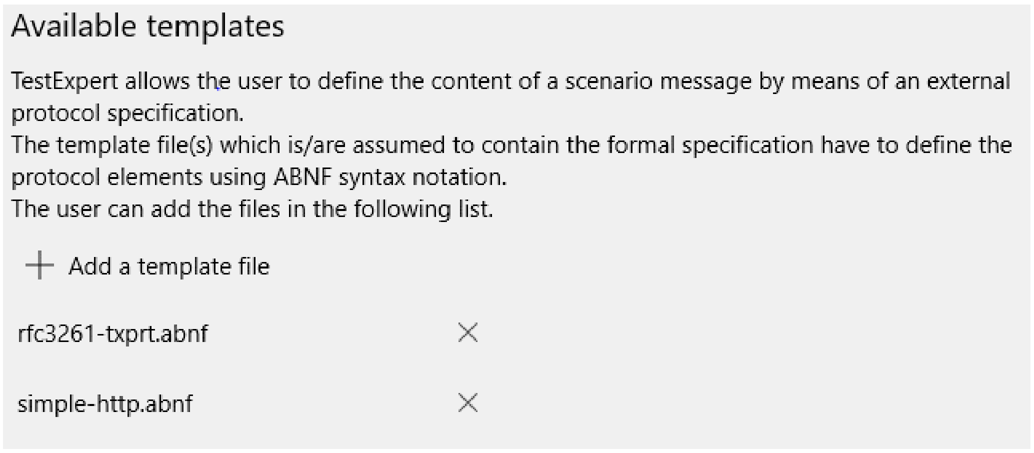


You can then configure the following properties:

Property	Description
Templates	Open the *Templates* view to manage possible message template files you want to use.
Message format	Open the *Message format* view to configure a couple of message formats such as big or little endian, UTF8/ASCII.
Communication	Open the *Communication* view to configure the communication channel(s) that you want to make use of when connecting with the system under test. This includes both outgoing (target) and listening channels.

Templates

Template files can be used for defining the content of text based protocol messages. If you want to use them then you first have to add the file to the solution. Do this by opening the *Templates* view and then tapping the *Add a template file* option.



Available templates

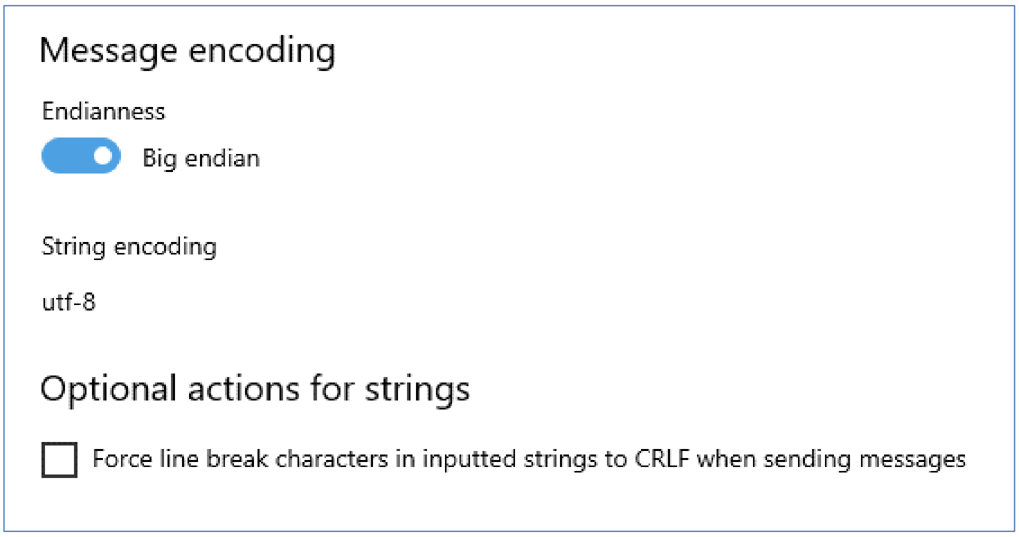
TestExpert allows the user to define the content of a scenario message by means of an external protocol specification.
The template file(s) which is/are assumed to contain the formal specification have to define the protocol elements using ABNF syntax notation.
The user can add the files in the following list.

The *Open* dialog box will appear. Locate and select your template file, then click *Open*.

Note: Template files are assumed to be text files containing ABNF message definitions. TestExpert assumes that these files have a type *.abnf* but you are free to give them any name and/or type. Refer to the ABNF template files chapter for a description of the file format. Also note there that TestExpert makes use of some proprietary 'prose-val' definitions to assist the user when editing single line and multi-line text using the ABNF message editor.

Message format

The *Message format* view lets you configure a couple of *Raw* message encoding options.


Message encoding

Endianness

Big endian

String encoding

utf-8

Optional actions for strings

☐ Force line break characters in inputted strings to CRLF when sending messages

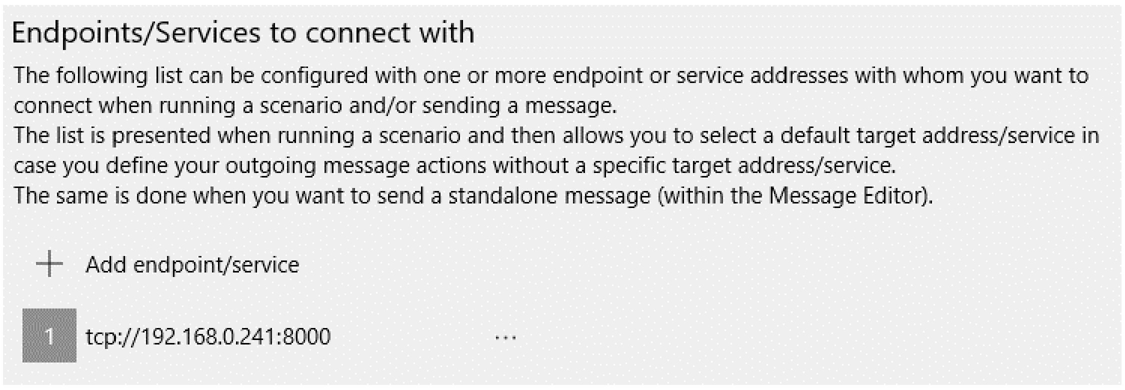
Encoding	**Description**
Endianness	You can select *Big endian* or *Little endian*. This option only applies to messages that are defined using TestExpert's *Raw* message syntax. That syntax supports definitions in terms of short, integer, and long values. The selected endianness then defines how the 2, 4 or 8 bytes of those values are ordered 'on the wire'.
String encoding	Currently only UTF-8 (i.e. 8 bit ASCII) is supported.
Force line break	You can check 'Force line break to CRLF..' if you want the app to replace single newline characters (\r and \n) by \r\n (CRLF) in a multi line text box. This is supported for ABNF type of messages in all message definition panels where a text box is presented to enter content. The default action with a multi line text box when pressing the Enter key is to insert a single \n character to create a new line.

Communication

When you tap the *Communication* view you can configure target endpoints/services and listener endpoints/services.

Target endpoints and Services

The top section lets you configure one or more communication endpoints or services that TestExpert will use to exchange messages. These are 'target' endpoints/services which means that TestExpert will take the initiative to establish the connection (a communication channel) when needed.



Endpoints/Services to connect with

The following list can be configured with one or more endpoint or service addresses with whom you want to connect when running a scenario and/or sending a message.
The list is presented when running a scenario and then allows you to select a default target address/service in case you define your outgoing message actions without a specific target address/service.
The same is done when you want to send a standalone message (within the Message Editor).

You can configure as many target endpoints/services as you want. When you run a scenario you will have the possibility to select which of the configured endpoints or services will be used for setting up the so-called default communication channel. That channel will then be used by TestExpert when it must send a message for which no specific endpoint has been assigned.

To add an endpoint or a service tap the *Add endpoint/service* button. You will then get a popup where you can choose the type of transport protocol, the host address and port, and other transport-specific properties.

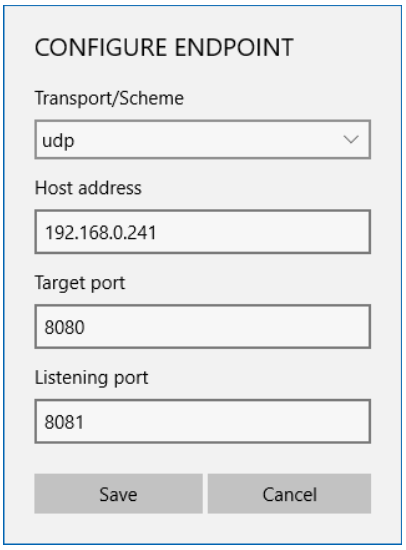
TestExpert supports the following transport protocols for target endpoints:

- udp
- tcp
- http(s)
- http(s) post

The following subsections describe these options in more detail.

UDP target endpoint properties

Select *udp* if you want to have (or forced to have) a low level datagram transport protocol. It is the fastest but connectionless and thus in essence unreliable.

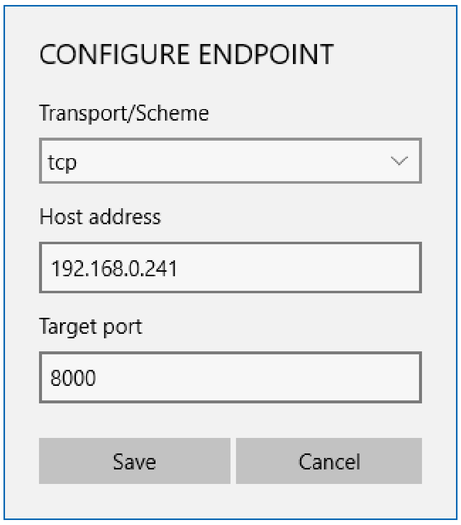
Property	**Description**	**Properties view**
Transport/scheme	Select *udp.*	 CONFIGURE ENDPOINT Transport/Scheme: udp Host address: 192.168.0.241 Target port: 8080 Listening port: 8081
Host address	You must provide here the IP address of the target device you want to connect with. Hostnames are not allowed.	
Interface to listen on	Tap the dropdown box to select the interface on which you want to bind the listener service. Select *All interfaces* to tell TestExpert to accept connections occurring on any of the installed interfaces.	
Target port	Enter the target/destination port number.	
Listening port	Enter the listener port number.	

Note:

For a UDP target endpoint you must specify both the target port and the listening port. The latter is used to receive remote messages that are triggered by the targeted device. As is common with UDP the listening port number will be passed as source port when TestExpert sends a UDP datagram to the endpoint.

TCP target endpoint properties

Select *tcp* if you want a low level and reliable protocol that is fast and reliable.

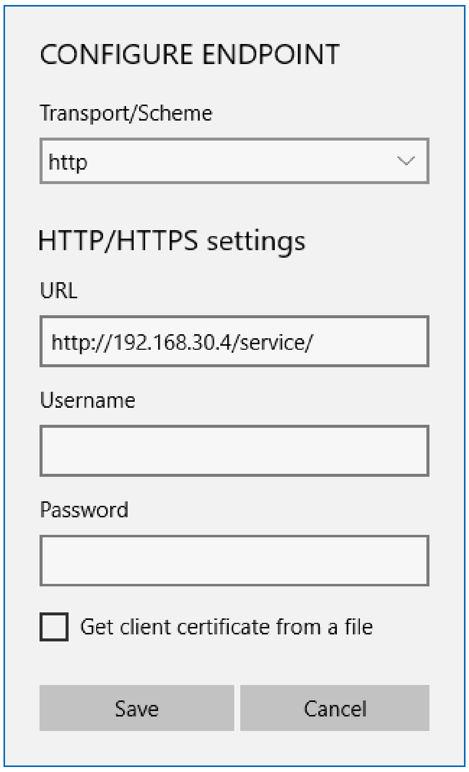
Property	**Description**	**Properties view**
Transport/Scheme	Select *tcp*.	 CONFIGURE ENDPOINT Transport/Scheme: tcp Host address: 192.168.0.241 Target port: 8000
Host address	You must provide here the IP address of the target device you want to connect with. Hostnames are not allowed.	
Interface to listen on	Tap the dropdown to select the interface on which you want to bind the listener service. Select *All interfaces* to tell TestExpert to accept connections occurring on any of the installed interfaces.	
Target port	Enter the target/destination port number.	
Listening port	Enter the listener port number.	

Note:

TCP, being a stream protocol, normally requires that some form of message framing data is present in outgoing messages. Various methods are around to accomplish this such as: a count indicator at some fixed position (e.g. at the start of the message), a tail marker (e.g. a CR in text-based messages), a length indicator in some kind of header block (e.g. something like the Content-Length header in HTTP), etc. TestExpert doesn't take care of this so the user must make sure that the necessary framing data is specified in the message content itself.

HTTP/HTTPS target endpoint properties

Select *http* if you want TestExpert to send HTTP request messages to the configured endpoint and receive HTTP response messages from the endpoint.

Property	**Description**	**Properties view**
Transport/Scheme	Select *http*.	
URL	Enter the URL path here which TestExpert will use when sending a HTTP request message. Note that the URL path must start with http:// or https://. Example: http://192.168.30.4/service	
Username	Enter the username when the target service expects user authentication.	
Password	Specify the password when the target service expects user authentication.	
Get client certificate from a file	Not supported yet.	

CONFIGURE ENDPOINT

Transport/Scheme: http

HTTP/HTTPS settings

URL: http://192.168.30.4/service/

Username:

Password:

☐ Get client certificate from a file

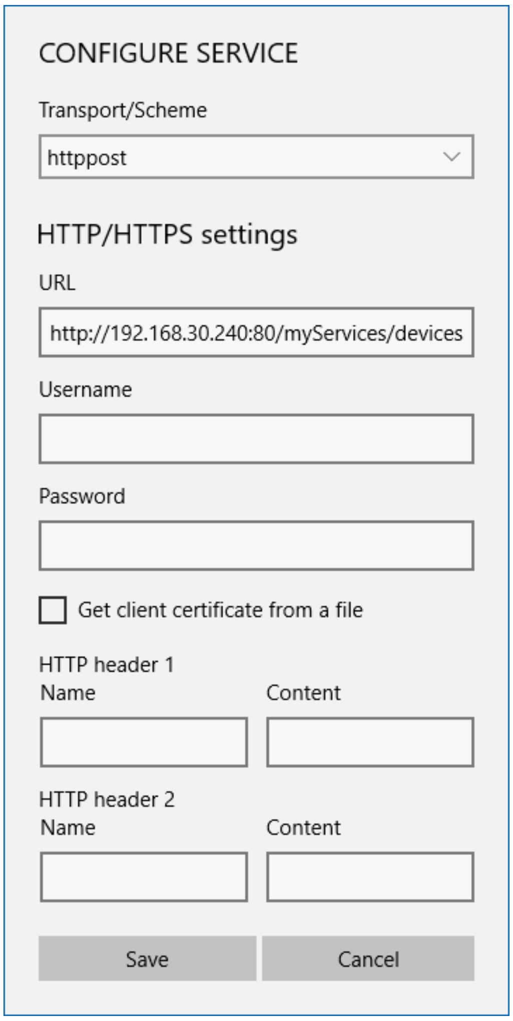
Notes:

1. Secure HTTPS will automatically be enabled when the URL starts with https.
2. TestExpert will only allow this type of endpoint to be used for messages that have been created by means of the HTTP Message Editor.
3. Chunked data transfer is not supported.

HTTP/HTTPS POST target endpoint properties

Select *httppost* if you want to send messages using a HTTP POST request and expect replies from HTTP response messages. TestExpert will then pass all message data that must be sent to the endpoint as content data of a HTTP POST message. The target service is expected to return the response data in a HTTP 200 OK response message.

Note: Select this type of transport for a target endpoint when you want to interact with a target device on which the Remote Method Invocation Connector is running.

Property	**Description**	**Properties view**
Transport/scheme	Select *httppost.*	 CONFIGURE SERVICE Transport/Scheme: httppost HTTP/HTTPS settings URL: http://192.168.30.240:80/myServices/devices Username: Password: ☐ Get client certificate from a file HTTP header 1 Name: Content: HTTP header 2 Name: Content: Save Cancel
URL	Enter the URL path here which TestExpert will use as request URI in the HTTP POST message. Note that the URL path must start with http:// or https://. Example: `http://192.168.30.2/rmic`	
Username	Enter the username when the target service expects user authentication.	
Password	Specify the password when the target service expects user authentication.	
Get client certificate from a file	Not supported yet.	
HTTP header 1 *HTTP header 2*	You can define up to 2 HTTP headers which TestExpert will include in addition to the standard headers.	

HTTP POST messages

TestExpert will issue all POST messages using HTTP/1.1.

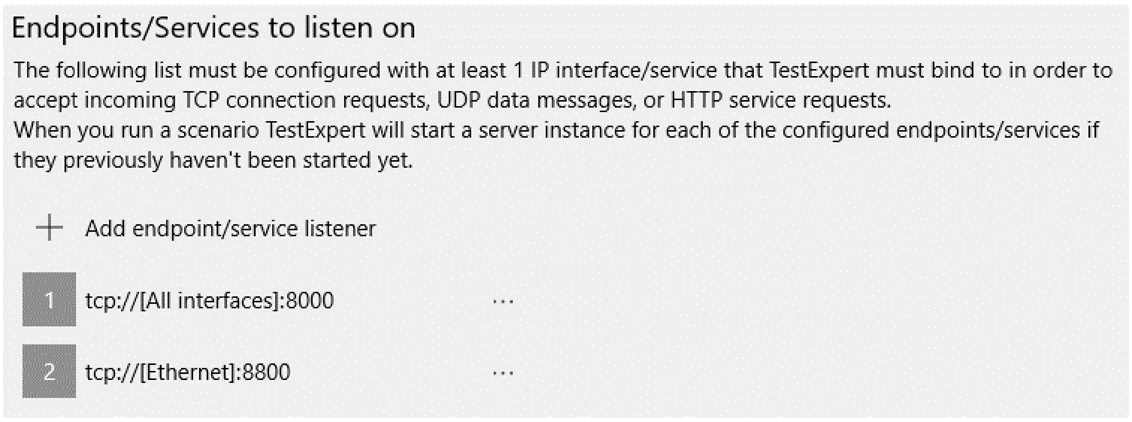
The following headers will automatically be included:

```
Accept-Encoding: gzip, deflate
Host: <hostname-of-the-requested-uri>
```

```
Connection: Keep-Alive
Cache-Control: no-cache
Content-Type: <type-string-for-the-included-content>
Content-Length: <length of the content>
```

Listener endpoints and Services

The bottom section lets you configure one or more communication endpoints or services for which TestExpert must bind a Server. These are ‘listener’ endpoints/services which means that TestExpert will simply start a server function for them when told so in the scenario and then will have to wait for incoming connections to appear. For these endpoints and services TestExpert will only be able to send messages as a reaction/response to a received message.


Endpoints/Services to listen on

The following list must be configured with at least 1 IP interface/service that TestExpert must bind to in order to accept incoming TCP connection requests, UDP data messages, or HTTP service requests.
When you run a scenario TestExpert will start a server instance for each of the configured endpoints/services if they previously haven't been started yet.

1 tcp://[All interfaces]:8000 ···

2 tcp://[Ethernet]:8800 ···

To add a listener endpoint or service tap the *Add endpoint/service listener* button. You will get a popup where you can choose the type of transport protocol, the interface to listen on, the port number and other transport-specific properties.

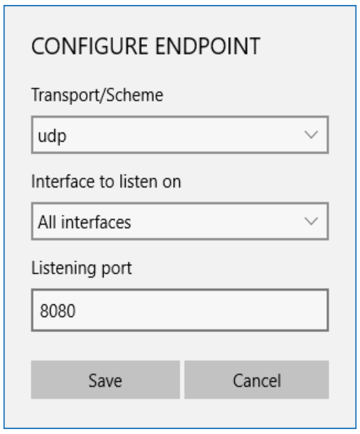
TestExpert supports the following transport protocols for listener endpoints:

- udp
- tcp
- http

The following subsections describe these options in more detail.

UDP listener endpoint

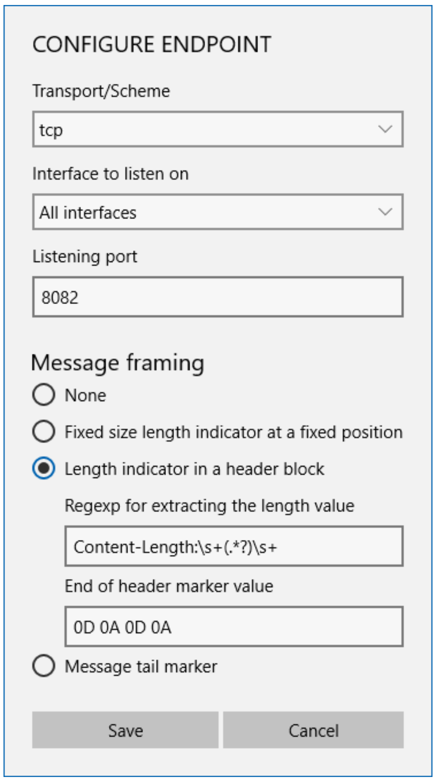
Select *UDP* if you want to have (or forced to have) a low level datagram transport protocol. It is the fastest but connectionless and thus in essence unreliable.

Property	Description	Properties view
Transport/scheme	Select *udp.*	 CONFIGURE ENDPOINT Transport/Scheme: udp Interface to listen on: All interfaces Listening port: 8080 Save Cancel
Interface to listen on	Tap the dropdown box to select the interface on which you want to bind the listener service. Select *All interfaces* to tell TestExpert to accept connections occurring on any of the installed interfaces.	
Listening port	Enter the listener port number.	

Note: With a UDP listener endpoint TestExpert will be able to accept incoming UDP messages from any device you want. When responding TestExpert will send a UDP datagram message to the IP address that originated the message and with the destination port set to the originator's source port.

TCP listener endpoint

Select *TCP* if you want a low level and reliable protocol that is fast and reliable.

Property	Description	Properties view
Transport/Scheme	Select *tcp*.	 CONFIGURE ENDPOINT Transport/Scheme: tcp Interface to listen on: All interfaces Listening port: 8082 Message framing None Fixed size length indicator at a fixed position [✓ Length indicator in a header block] Regexp for extracting the length value: Content-Length:\s+(.*?)\s+ End of header marker value: 0D 0A 0D 0A Message tail marker Save Cancel
Interface to listen on	Tap the dropdown to select the interface on which you want to bind the listener service. Select *All interfaces* to tell TestExpert to accept connections occurring on any of the installed interfaces.	
Listening port	Enter the listening port number.	
Message framing	Select which of the 4 possible message framing methods TestExpert must use to detect whether a received TCP message is complete. See the next Message framing section for a description.	

Message framing

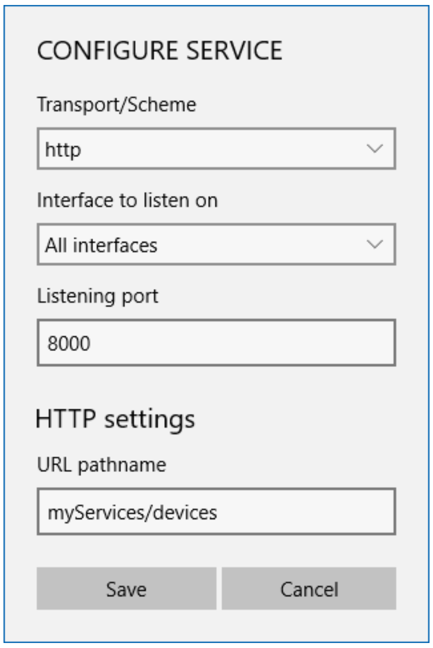
Because TCP is a stream protocol TestExpert needs to be told what type of message framing is expected. If it doesn't know that it will be impossible to figure out whether a complete message has been received.
There is no standard available in the communication protocol world for people on how to do the message framing. As a consequence various implementations exist: some have a length prefix, some use a tail marker, some others carry a named length field, etc. It is impossible for TestExpert to support everything that is out there in the world, but at least the most frequently used methods are supported. These are:

- **None** – Choose this if none of the next 3 methods are suitable and you are sure that messages are not sent/received in chunks. That will work out very often (but not always) when TestExpert and the device under test are all part of the same local network.
- **Fixed size length** – Choose this when your messages have a length indicator at a fixed position in the message whereby the length is stored as a binary value in a fixed number of octets. You must define:
  - The *position*, i.e. the offset in the message where the length starts (with 0 being the first octet).
  - The *number of octets* that is used to hold the length value. If the length spans multiple octets then TestExpert will assume that the length is stored in big endian format.
  - A possible *adjustment* value that TestExpert must add to the length. If you want TestExpert to subtract this value from the retrieved length then specify the value using a minus sign, like in:
    `-1`
- **Length indicator in a header block** – Choose this method when you have a text message where the length is defined as a 'header' value, like with the Content-Length header in HTTP messages. In this case you must define 2 additional properties:
  - A *regexpr* (regular expression) that allows TestExpert to both find the length 'header' and extract the assigned length value. So you must make sure that the regular expression not only can find the header but also defines a sub-pattern inside a pair of parenthesis to capture the length as a separate group. TestExpert assumes that the last captured group holds the length.
    Example: Assume that all messages carry a 'Content-Length' header (like in HTTP). The regexp will then be:
    `Content-Length:\s+(.*?)\s+`
  - The *End of header marker* that is expected to be present in any received message and which marks the end of all possible headers. The marker must be defined as being 1 or more octets where each octet is specified by means of 2 hexadecimal characters (case insensitive).
    Example: Assume that each header ends with a CR and LF character and there is an additional CR and LF character after the last header. The end marker must then be defined as:
    `0D 0A 0D 0A`
- **Tail marker** – Choose this option when the end of the message is defined by one or more unique octets. You must define each of the octets by means of 2

hexadecimal characters (case insensitive), like in:
`FF FF FF FF`
Note: With this method the sender must make sure that the tail marker is unique and therefore content that is the same as the tail marker must be escaped in some way. TestExpert doesn't deal with this.

HTTP

Select *HTTP* if you have test scenario's where TestExpert must react as a HTTP server; i.e wait for HTTP request messages and respond with HTTP response messages.

Property	**Description**	**Properties view**
Transport/scheme	Select *http.*	 CONFIGURE SERVICE Transport/Scheme: http Interface to listen on: All interfaces Listening port: 8000 HTTP settings URL pathname: myServices/devices
Interface to listen on	Tap the dropdown to select the interface on which you want to bind the listener service. Select *All interfaces* to tell TestExpert to accept connections occurring on any of the installed interfaces.	
Listening port	Enter the listening port number.	
URL pathname	Enter a pathname here which a client device must pass in the URI part of its HTTP request towards TestExpert. You can define both partially and fully specified pathnames. A partially specified pathname contains a trailing '*' wildcard character. Only requests with a matching pathname will be passed on to a running scenario.	

TestExpert's HTTP Server/Listener

When a HTTP listener endpoint is started in a scenario TestExpert will instantiate a small HTTP server instance. It will accept any incoming HTTP request using the following checks:

- The method must be: GET, PUT, POST, DELETE, HEAD, OPTIONS, or PATCH.
- The URL pathname must match the configured endpoint's URL.
- HTTP headers must have the correct format.
- Multipart content must contain correctly formatted boundary entities.

The server will never by itself respond to an incoming request except when the above checks fail. In that case the server will respond with a 400 Bad Request response.

The service will use the Content-Length header in the incoming request to find out when a message is complete. When so, the request is passed on to the running scenario which typically must react by responding with a HTTP response message.

Chunked data transfer is not supported.

Reference information

This part of the user's guide covers the following topics:

Topic	Description
Message description language	A detailed description of the message description language that is available to define the payload of messages and message information elements.
ABNF template files	An explanation about the syntax of ABNF files.

Message description language

To allow the user to define different kind of messages and information elements, TestExpert provides a small, interpreted *Message Description Language*. You will need to know the syntax of this language if you want to define the different data fields of incoming and outgoing messages when you want to define those messages in so called 'raw' format.

E.g., assume you want to define a 'raw' outgoing message, comprising of 3 fields.

- The first field consists of a 1 byte message type.
- The second field consists of a 1 byte field type, a 2 byte length value (holding the length of the field 'data'), and a variable length data part.
- The third field has the same layout as the second.

TestExpert's *Message Description Language* allows you to define such a message as follows:

Field	Definition
Field 1: Message type	`0x01`
Field 2: type	`0x07`
Field 2: Length	`@length("Device")`
Field 2: name	`"Device"`
Field 3: type	`0x11`
Field 3: length	`@length("Smith")`
Field 3: name	`"Smith"`

The payload data of the above message will be:

```
Content:

000   01 07 00 00 00 06 44 65   ......De
008   76 69 63 65 11 00 00 00   vice....
010   05 53 6D 69 74 68          .Smith
```

TestExpert provides a small set of predefined data types, operators to manipulate those types, and a set of statements and functions for performing control. You will make use of them when defining the contents of incoming and outgoing messages and when specifying conditions on scenario actions.

Data types

TestExpert's *Message Description Language* supports the following set of data types for representing values:

- The type *byte* is used to represent a value occupying 8 bits (1 octet). It is the 'smallest' type.
- The type *word* is used to represent a value occupying 16 bits (2 octets). When used on messages, you can choose to use the big endian or little endian format notation.
- Type *integer* is used to represent a value occupying 32 bits (4 octets). When used on messages, you can also choose to use the big endian or little endian format notation.
- Type *long* is used to represent a value occupying 64 bits (8 octets). When used on messages, you can also choose to use the big endian or little endian format notation.
- Type *string* is used to represent a set of characters. The number of bytes that each character takes is configurable; i.e. when the string encoding is set to ‘utf8’ then each character will occupy 1 byte.
- Type *set* is used to group a set of data values, expressions or functions together and treat them all as a single unit.

There are a number of ways how you can specify values of the above listed data types:

- Literals
- Strings
- Compound blocks
- @Statements
- Expressions
- Named variables

Literals

Using literals you simple specify a value as either a *byte*, *word*, *integer* or a *long* value. Values are entered in 2 ways:

- In hexadecimal, using the C/C++/C# way of defining hexadecimal values. Hexadecimal bytes are therefore entered by prepending `0x` or `0X` to a literal integer. Both uppercase and lowercase 'A' to 'F' characters can be used.
- In decimal notation. Anything different from the '0x..' notation will be considered as being a decimal value.

For specifying *word, integer,* and *long* values, there is also a 3rd method which can force the given literal integer (having decimal notation) to be of a specific type by appending character 'w' or 'W', ‘i’ or ‘I’, ‘l’ or ‘L’ to the literal.

E.g.:

```
0x10        // hexadecimal byte
0x0010      // hexadecimal word
0x00000010  // hexadecimal long
100         // decimal byte
300         // decimal word
80w         // a word
1L          // a long
```

In case no type modifier ('w', 'W', 'i', 'I', 'l' or 'L') is specified, then the resulting type is either explicitly given (e.g. 0x0010 will be a *word*) or implicitly (e.g. 100 will be a *byte* because it can fit in 8 bits).

Strings

String values can be used to refer to a sequence of characters. They are entered using the 'double quote' character (") as heading and trailing character. E.g.:

```
"my string"
"another, longer string"
```

Compound blocks

Use a compound block to specify one or more values belonging to a set.

You can construct such a set by enclosing the expressions/statements between opening and closing brackets: `'{'` and `'}'` like in:

```
{0x10 "ABC" @var(ID)}
```

```
{0x20 0x21 0x45 "Fred" @element(connect, 0, 2560) 0x80}
```

TestExpert also allows you to define empty block statements: a '{' character, immediately followed by a '}' character. You might want to use such a construct with conditional statements to have either a set of data values or a null value.

@Statements

@Statements are reserved ‘functions’ in TestExpert that accomplish a specific purpose, like: get the value of a variable, get the byte at the current message parser position, etc.

All @statements start with a ‘@’ characters followed by a name, followed by an open parenthesis, followed by 0 or more parameters, followed by a closing parenthesis.

Examples:

```
@var(ID)    // Get the value of variable ‘ID’
@$(2)       // Get the 2 bytes at the current parser position
```

See Statements and functions for a detailed description.

TestExpert expressions

Expressions include nearly all of the usual programming language capabilities. They include the usual operators (! , * , / , + , - , & , | , == , < , <= , > , >= , |= , && , ||). You can use them to build expressions, like:

```
    (0x20 + 0x1000),
    (@var(_variable_name) / (100 * 2)),
    etc.
```

Similar to other 'programming languages', expressions in TestExpert's *Message Description Language* are composed of one or more operations. The evaluation of an expression performs one or more operations, yielding a result. E.g.:

```
    10 * @var(var1)
    @var(var1) / 20
```

You can specify all operations within an expression construct but remember that TestExpert evaluates the expression from left to right. If this is not what you want then you have to include opening and closing parenthesis characters. E.g.:

```
    20 + (10 * @var(var1))
```

The data type of the result (a *byte*, *word*, *integer*, *long*, or *string*) is in general determined by the data type of the operand(s). When more than one data type is present, type conversion takes place when possible whereby the resulting type will be the type of the operand with the 'biggest' type (a *long* is 'bigger' than a *word*, which is 'bigger' than a *byte*). E.g.:

```
   (0x11223344 >> 8) results in: 0x00112233
   (0x1122 >> 8) results in: 0x0011
```

You can however force the value of a numeric expression result to be a *byte*, *word*, *integer*, or *long* value by adding leading and trailing parenthesis characters and appending the `@byte`, `@word`, `@int`, or `@long` keyword to the expression. E.g. (assume var1 = 20):

```
    ((2 * 1024) + 0xFF000011)@byte, result: 0x11
    (10 + (@var(var1) | 0x80))@word, result: 0x9E
    (0x11223344 >> 8)@word, result: 0x2233
    ((@randomnum(100,255))@word, result: 0x0087
    (@var(var1) / 20)@long, result: 0x0000000000000001
```

The simplest form of an expression consists of a single literal constant, string, or variable. E.g.:

```
    0x200
    "abc"
```

When using operations in expressions, all operands must have *byte*, *word*, *integer*, or *long* as data type. The only 2 exceptions are:

1. A single decimal character string - Such a string is also allowed. E.g.:

```
    (0x10 + "1" + 0x20 ), results in a value: 0x61
    (0x10 + @left("123",1) + 0x20), results in a value: 0x61
    (0x10 + "123" + 0x20), results in an error
```

2. All operands are strings – In that case the strings are concatenated. E.g.:

```
    ("12" + "34"), results in a new string: "1234"
```

Arithmetic operations

Operations can be:

*	multiplication	expr * expr
/	division	expr /expr
%	modulus (remainder)	expr % expr
+	addition	expr + expr
-	subtraction	expr - expr

Equality, relational and logical operations

These operations evaluate to either true or false. A truth condition yields `1`; a false condition yields `0`.

!	logical NOT	!expr
<	less than	expr < expr
<=	less than or equal	expr <= expr
>	greater than	expr > expr
>=	greater than or equal	expr >= expr
==	equality	expr == expr
!=	inequality	expr != expr
&&	logical AND	expr && expr
||	logical OR	expr || expr

The logical NOT operator ("!") evaluates to true if its operand has a value of zero; otherwise it evaluates to false.

Bitwise operators

Bitwise operators allow the user to test and set individual bits or bit subsets. The operands of bitwise operators must be of an integral type.

~	bitwise NOT	~expr
<<	left shift	expr << count_expr
>>	right shift	expr <> count_expr
&	bitwise AND	expr & expr

^	bitwise XOR	expr ^ expr
|	bitwise OR	expr | expr

Named variables

TestExpert allows you to assign values to static variables and dynamic variables. You can get the value of those variables in 2 ways:

- Explicitly by referencing them using the @var() statement, like in:
  `(@var(var1) / 20)`
- Implicitly by referencing them directly using their name, like in:
  `(var1 / 20)`

The latter method only works when the use of the direct name is not in conflict with other elements in the expression.

Statements and functions

Data types and expressions are the main constructs of TestExpert's *Message Description Language* that allow the user to specify the internal bytes of incoming and outgoing messages. Expressions however lack the possibility to a more dynamic way of controlling the message settings.

Statements and functions will allow the user to have more control on how particular bytes have to be set in messages and on which bytes have to be set.

Statements

@$	Octet at the current position of the message parser.
@atoi()	Convert a string to a numeric value.
@element()	Insert the definition(s) of the specified message/information element.
@itoa()	Convert a numeric value to a string.
@jvalue()	Extract a value from a context string using a JSONPath expression.
@left()	Take leftmost part of string.
@length()	Return the length of a string.
@match()	Check if a given context string contains a user specified pattern.
@right()	Take rightmost part of a string.
@rvalue()	Extract a value from a context string using a regular expression.
@set()	Assign a value to a run-time variable.

@setparm()	Set the given parameter number.
@skip()	Skip octets.
@this.length()	Return the length of a message/element.
@this.parm()	Take the given parameter number.
@var()	Get the value of a static or runtime variable.
@xvalue()	Extract a value from a context string using an XPATH expression.
@randomstr()	Obtain a random string of a given length (characters are: a..zA..Z0..9).
@randomnum()	Obtain a random number between a given min an max value.

Statements syntax

Throughout the next pages the following special characters are used to define the statement syntax for defining message:

`[ ]`	Identifies an optional argument. Arguments not enclosed in brackets are required.
`...`	Indicates that you can specify multiple values for the previous argument.
|	Indicates mutually exclusive information. You can use the argument to the left of the separator or the argument to the right of the separator. You cannot use both arguments in a single use of the command.
`{ }`	Delimits a set of mutually exclusive arguments when one of the arguments is required. If the arguments are optional, they are enclosed in brackets ([ ]).

@$

The @$ indicator can only be used on incoming messages and represents 1 or more octets of the received message. The location of the octets is defined by the value of the incoming message parser position as it is at the moment that the `@$` statement is encountered.
You can use this indicator when you want to assign a *byte*, *word*, *integer, long* or *string* value from a received message to a run-time variable, or when you want to assign an array of bytes to a run-time variable.

Syntax

```
1. @$
```

This syntax represents the *byte* value of the one octet at the current parser position in the message.

```
2. @$(int count)
```

This syntax takes as many octets, starting from the current parser position, as indicated by parameter *count*. The expression must result in a positive numeric value.

When used in a `@set` statement without any type override, the value of the run-time variable is set to an array of bytes, that is filled with the octets from the message.

Example

Assume that the content of an incoming message is defined as follows:

Field	**Definition**
Assign first (unknown) byte to 'rcvdByte' and then skip it	`@set(rcvdByte, @$) @skip(1)`
Next data: "ABC"	`"ABC"`
Assign next 4 octets to 'rcvdInt' and then skip/ignore them	`@set(rcvdInt, @$(4), @int) @skip(4)`
Next octet is expected to hold a number; save it in ‘rcvdCount’	`@set(rcvdCount, @$)`
Skip the count octet	`@skip(1)`
Assign the next ‘rcvdCount’ number of octets to 'rcvdLong'	`@set(rcvdLong, @$(@var(rcvdCount), @long))`
Skip the octets	`@skip(@var(rcvdCount))`
Assign the next 6 octets to the 'rcvdData' array	`@set(rcvdData, @$(6)`
Then: skip these 6 octets	`@skip(6)`

Assuming an incoming message with the following contents:

```
0x10 0x41 0x42 0x43 0x07 0xCC 0x33 0x11
0x03 0x11 0x22 0x33 0x01 0x02 0x03 0x04
0x05 0x06
```

When applied against the above definition, the following run-time variables will be created:

- 'rcvdByte' - its value will be 0x10
- 'rcvdInt' - its value will be 0x07CC3311
- 'rcvdCount' - the value will be set to 0x03
- 'rcvdLong' - 3 octets will be taken to form 0x0000000000112233 as a *long* value
- 'rvcdData' - this is an array with contents: 0x01 0x02 0x03 0x04 0x05 0x06

@set

The `@set` statement allows you to initialize a given run-time variable with an appropriate value. In case the run-time variable hasn't been created before TestExpert will create it.

Syntax

```
@set(string name, mdlDataType setValue[, typeOverride])
```

Parameter *name* defines the name of the runtime variable. It can be any new or already existing name in the list of run-time variables.

Parameter *setValue* defines the value that must be assigned to the runtime variable. The resulting value can be: a byte, a word, an integer, a long, a string, a compound block, or an array of bytes.

Parameter *typeOverride* is optional. Acceptable type overrides are: *@word*, *@int*, *@long*, and *@string*. When provided the value that results from *setValue* will be converted to the indicated type. With the conversion to *word*, *integer*, or *long* TestExpert takes the configured endianness into account.

Example

Assume the following incoming message definition:

field	**definition**
field1	`"12345"`
field2: 0x84	`0x84`
field3: 32 bit ID	`@set(id, @$(4), @int) @skip(4)`
field4: 0x44	`0x44`
Field5: 32 bit KEY	`@set(key, @$(4)) @skip(4)`
other (if any)	`@skip(*)`

When the specified message is being received, then 2 runtime variables will be created: ‘id’ and ‘key’.
Note the difference between the @set statement in field 3 and 5. Both statements save 4 bytes. However, the first statement really saves the 4 bytes in *integer* format (e.g. 0x00001122) whereas the second statement saves the 4 bytes one after each other (e.g. 0x11 0x22 0x33 0x44).

@element

The `@element` statement allows you to insert a Message element into a message that is defined using the Raw message editor or into another *Message element*. It is an essential feature allowing you to reuse predefined element specifications within multiple scenario messages.

When you insert a message element you actually only insert a reference to the element at a given location of a message or an element of a message. TestExpert will do the real insertion at the moment an outgoing message must be sent out, or an incoming message must be verified, or when you want to check whether a message definition is correct.

Compared with filling out all the message contents yourself, importing has the advantage that you can reuse message definitions which are global to a given project. This is the recommended way of working because it will allow you to manage common messages and items at one place for different scenarios.

Syntax

```
@element(string pathName[, mdlDataType parm]...)
```

Insert the message element indicated by *pathName*. This is a folder-like path string formatted as follows:

```
    "groupName[/folderName]/elementName"
```

Where:

- *groupName* – One of the 2 supported message elements groups.
    1. *ome* - This group contains all outgoing message elements. You will store here all the reusable message elements that you want to import in other outgoing elements or messages.
    2. *ime* - This group contains all incoming message elements. You will store here all the reusable message elements that you want to import in other incoming elements or messages.
- *folderName* - One or more folder names defining the parent(s) of the element.
- *elementName* - The name of the referenced element.

In case you have to deal with a parameterized element then the statement lets you specify as many *parm* values as necessary. You can make use of any of the supported value definitions when passing a parameter; i.e. literals, strings, @statements, compound blocks, and expressions.

Example 1 - Simple

The simplest way how a user would specify this in a message, looks as follows:

Field	**Definition**
field1	`0x22`
Include staticUserName	`@element("ome/staticUserName")`
next field	`"1234"`

When you run the scenario, or verify the message, TestExpert will read all the definitions from the referenced element and put them in the output message behind possible other definitions that have been specified using the normal straight syntax.

E.g.: Assume that 'staticUserName' in the above example defines an outgoing information element that looks as follows:

Field	**Definition**
Byte	`0x01`

String	`"John.Doe"`
End marker	`0x00`

The above example would then produce the following message content:

```
000  22 01 4A 6F 68 6E 2E 44      " John.D
008  6F 65 00 31 32 33 34         oe 1234
```

Example 2 – Parameterized elements

Assume we have 2 parameterized information elements: 'userName' and ‘userData’. They look as follows:

userName:

Field	**Definition**
Field type	`0x01`
String	`@this.parm(1)`
End marker	`0x00`

userData:

Field	**Definition**
Field type	`0x02`
User data length	`@len(@this.parm(1))`
User data	`@this.parm(1)`

Both elements are included in the following message:

Field	**Definition**
field1	`0x22`
import a info field	`@element("ome/userName", "Pete.Smith")`
next field	`"1234"`
import with a block parameter	`@element("ome/userData", {0x10 0x77})`
End of message	`0xFF`

The payload of this messages will then be:

```
000  22 01 50 65 74 65 2E 53      " Pete.S
008  6D 69 74 68 00 31 32 33      mith 123
010  34                           4
```

@this.parm

The `@this.parm` statement can be used inside *Message elements* to get the value of a parameter that is assumed to have been set in the message/element that includes this element.

Syntax

```
@this.parm(int parmNumber)
```

The statement requires an argument *parmNumber* that indicates which of the possible multiple parameters has to be taken. Parameter numbers can be: 1, 2, etc.

parmNumber can be a fixed number, a `@var` value, or an expression that results in a non-zero, positive numeric value.

Example

Assume the following message element, called ‘2Parms’

Field	**Definition**
field1	`0x22`
field2 (input argument 1)	`@this.parm(1)`
field3	`0x1122`
field4 (input argument 2)	`(0x8000 + @this.parm(2))`
field6	`0xFF 0x00`

The message that includes the above information element is defined as follows:

field	**definition**
Include 2Parms element	`@element("ome/2Parms", "ABC", (20 + 10))`

The resulting payload of the message will be (number encoding: big endian):

```
000  22 41 42 43 11 22 80 1E      "ABC "
008  FF 00
```

@this.length

The `@this.length` statement gets the length in bytes of the message or the element within which the statement appears.

You can use the statement both in an expression and to put the length within the message/element itself. In the latter case the length is stored as a byte value. If you want it different then you must add one of the type override statements: *@word*, *@int*, or *@long*.

Syntax

`@this.length()`

The statement doesn't require any argument.

Example

Assume the following outgoing message element, called 'Section02'

Field	Definition
Section id	`0x02`
Section length	`@this.length()`
field1	`0x22`
Field2	`0x1122`

The message that includes the above information element is defined as follows:

field	definition
Length	`(@this.length())@word`
Include Section02 element	`@element("ome/Section02")`
Tail marker	`0xFF`

The resulting payload of the message will be (number encoding: big endian):

```
000  00 08 02 05 22 11 22 FF
```

@setparm

The `@setparm()` statement allows you to specify an argument that is to be passed through when importing contents from a message information element using the `@element()` statement. The imported message will reference such argument by means of the `@this.parm()` statement.

This statement is an alternative method for setting a parameter. The usual way to pass a parameter is to specify its value with the `@element` statement itself.

Syntax

`@setparm(int argumentNumber, mdlDataType value)`

You can set the value of a given input argument by specifying the `@setparm` keyword, followed by an *argumentNumber* and the *value* of the parameter between parenthesis.

ArgumentNumber must be a non-zero positive numeric value and denotes the 1st, the 2nd, etc. parameter of the element to be included.

You can make use of any of the supported value definitions when passing *value*; i.e. literals, strings, @statements, compound blocks, and expressions.

Example

Assume a message element called 'ie_with_3_arguments' that defines the following content:

field	**definition**
field1	`0x22`
field2 (input argument 1)	`@this.parm(1)`
field3	`0x1122`
field4 (input argument 3)	`@this.parm(3)`
field5 (input argument 2)	`@this.parm(2)`
field6	`"ieEND"`

Take then the following definition for a message 'MSG':

field	**definition**
message id	`0x2000`
	`@setparm(1, "ABC")`
	`@setparm(2, 0)`
	`@setparm(3, "abc")`
import info element	`@element(ie_with_3_arguments)`

This will result in the following content being defined for 'MSG':

```
Content:

000  20 00 22 41 42 43 11 22   ·"ABC·"
008  61 62 63 00 69 65 45 4E  abc·ieEN
010  44                       D
```

@skip

The `@skip` statement, typically used when defining incoming messages or information elements, allows you to ignore (skip) as many octets in the message as given by the expression which is given as parameter of the statement. You will use this when you don't know beforehand the actual values in a given part of an incoming messages and therefore want to tell TestExpert to ignore these values.

Syntax

```
@skip([int count|*])
```

The statement begins with the `@skip` keyword, followed by parameter *count* in parenthesis which indicates how many bytes have to be skipped.

In case you don't provide a value, then the count will be set to 1. E.g.: `@skip()`.

The `@skip(*)` statement can be used in the definition of an incoming message to tell the system that any further bytes in the parsed message must not be looked at anymore.
It is typically used when only the heading information fields of an incoming message must match with a given definition whereas the contents of the remaining, trailing fields is not important.

When specified, the `@skip(*)` statement must be the last statement in the message/element definition. Note however that the provision of this statement in a pattern message doesn't necessarily means that the validated message always needs to have one or more bytes at the given point that have to be ignored. The message can very well stop at the field that precedes the `@skip(*)` field. The `@skip(*)` statement skips the remainder of the message if there are any message bytes left.

Example

Assume a received message, starting with a 5 character string `"begin"`, followed by 5 undefined octets, followed again by a 3 character string `"end"`. An incoming TestExpert message that matches such a message would be the following:

field	**definition**
string 1	`"begin"`
5 undefined octets	`@skip(5)`
string 2	`"end"`

@length

This statement returns the length of a given item. The length is returned as a numeric value of type *integer* (4 bytes), but you can change that by adding one of the supported 'type-override' statements: *@byte*, *@word*, *@long*.

The calculated length refers to the number of octets that the item will have when used in the payload of a message. In case the item refers to a string then the length of the string will be the number of characters in the string.

Syntax

```
@length(mdlDataType item)
```

The input parameter *item* can be any type of item: a literal, a string, a compound block, a value returned by a @statement, or an expression.

Examples

Field	Definition
A string. Length: 4	`@length("1234")`
A block. Length: 7	`@length({0x01 "1234" 0x1122})`
A variable. (myVariable = 0x1020 0x30) Length: 3	`@length(myVariable)`
With type override Length: 0x0003	`@length(myVariable)@word`

@left

The @left statement returns a string containing a specified number of characters from the left side of a string.

Syntax

```
@left(string inpStr, int count)
```

@right

The @right statement returns a string containing a specified number of characters from the right side of a string.

Syntax

```
@right(string inpStr, int count)
```

@var

The @var statement returns the value of a specified variable. When TestExpert encounters this statement it tries to locate the variable in the list of *Runtime variables*. If it isn't there then the list of *Static variables* is looked up.

Syntax

```
@var(string varName)
```

Example

```
@var(rcvdCount)
```

Note: You can also write the name of the variable between double quotes like in:
@var("rcvdCount")

@atoi

The @atoi statement parses a string, interpreting its content as an integral number, and returns a value of type *integer*.

Syntax

```
@atoi(string inpStr)
```

@itoa

The `@itoa` statement converts an integer to a string.

Syntax

```
@itoa(int inpValue)
```

@match

This statement checks if there is a match between a user defined pattern and a given context string. It is primarily used in incoming message payload definitions to check if the character(s) at a given position in the payload of a received message can be accepted as being valid.

Syntax

```
@match(string regexPattern)
```

Parameter *patternStr* is a character/pattern string that contains a regular expression.

Example 1

```
@match(".*(?=\r)")
```

Matches all characters until a CR (0x0D) character is encountered.

Example 2

In case you want to have a double quote character in your regular expression value don’t forget to escape the double quote, like in:

`@match("'|\"")`

This matches a single quote character or a double quote character.

@rvalue

This statement uses a regular expression pattern to extract one or more parts from a context specific input string (e.g. a string or substring that sits in the content part of an incoming text message).

Syntax

`@rvalue(string regexPattern)`

Parameter *regexPatternStr* is a regular expression pattern that is used to find one or more matching parts. You can define patterns with or without “capturing groups”.

- In the first case the first matching string is extracted.
- With 1 or more capturing groups (each group is specified as a sub-pattern between a pair of parenthesis) the string captured in the last group is extracted.

Note: TestExpert uses the .NET regex engine.

Examples

Assume an input string containing `"100 Bananas".`
The statement `@rvalue("[0-9]+")` then extracts: `"100"`.

Assume an input string containing `"From: john@marshal.com".`
The statement `@rvalue("From:\s(.*)@(.*)")` then extracts: `"marshal.com"`.

@xvalue

The `@xvalue` statement extracts user-defined data from a context specific XML string (e.g. the XML document that sits in the content part of an incoming text message).

Syntax

`@xvalue(string xPath)`

Parameter *xPathStr* contains an XML Path Language (XPath) expression. Such expression uses a path notation, like those used in URLs, for addressing parts of an XML document. The expression is evaluated to yield a single string value (if there is only 1 result) or block value containing multiple strings (if there are multiple matches). For example, the expression `book/author` will return a block of strings of the

`<author>` elements contained in the `<book>` elements, if such elements are declared in the source XML document.
In addition, an XPath expression can have predicates (filter expressions) or function calls. For example, the expression `book[@type="Fiction"]` refers to the `<book>` elements whose `type` attribute is set to `"Fiction"`.

Examples

```
@xvalue("/bookstore/book[1]/title")
```

Returns the title of the first book in the bookstore. Note: Indexes start at 1.

```
@xvalue("/bookstore/book[@var(bookIndex)]/title")
```

Returns the title of the book that is located in the bookstore at the index defined by variable ‘bookIndex’.

```
@xvalue("/bookstore/book/title")
```

Returns a block of strings containing all the book titles.

@jvalue

The `@jvalue` statement extracts user-specified data from a context specific string that contains a JSON structure.

Syntax

```
@jvalue(string jsonPath)
```

Parameter *jsonPathStr* contains a JSONPath expression string. Such expression looks like XPATH but use the dot-notation and/or bracket-notation for input paths.

Example

```
@jvalue("$.store.book[" + @var(bookIndex) + "].title")
```

@randomstr

The `@randomstr` statement generates a random string with a length that is defined as argument. The string contains a combination of upper and lowercase A..Z characters and numeric characters 0..9.

Syntax

```
@randomstr(int length)
```

Parameter *length* specifies the length of the random string that is being returned.

Example

```
@randomstr(5) - Returns "Oij1K"
```

@randomnum

The `@randomnum` statement produces a random number that lies within a specified range. The statement returns the number as a 32 bit value.

Syntax

```
@randomstr(int minValue, int maxValue)
```

Parameter *minValue* specifies the inclusive lower bound of the produced number.

Parameter *maxValue* specifies the exclusive upper bound of the produced number. It must be greater than or equal to *minValue* and smaller then `2147483647.`

Example

```
@randomnum(5, 1000) - Returns 0x002000fA
(@randomnum(5, 500))@word - Returns 0x015C
```

ABNF template files

A correctly formatted ABNF file

There are some things you should know about what constitutes a valid ABNF file.

- TestExpert's ABNF file parser is compliant with the rule definitions and encoding that is laid out in RFC5234.
- Comments start with a semicolon «;».
- When the semicolon is at the start of a line, is followed by one or more whitespace characters, and then starts with a «!» character, then this will be interpreted as an option for the TestExpert ABNF message editor. Options are typically located at the beginning of the file.
- When the semicolon is not positioned at the start of a line (e.g. after a rule definition or indented on the next line) then the comment will be attached to the rule that precedes the semicolon.

ABNF message editor options

`!show(RuleName1,RuleName2, ..)`

When you include this option in the abnf file then the ABNF message editor will limit the set of rules that you can select, to only those defined in the list of rule names.

Assume the following abnf template file:

```
; !name("SimpleIOT simple-iot.abnf")
; !syntax("abnf")
; !import("core-abnf.abnf")
; !show(request,response)

request = (switch-on / switch-off / "Status" / "Reset") EOM
response = (ok / "INVALID") EOM
switch-on = "SwitchOn" ":" OWS device-id
switch-off = "SwitchOff" ":" OWS  device-id
device-id = <?>   ; Valid id's: 0..9
ok = ("OK" [CRLF message-body])
message-body = <??> ; The message body (if any) is used to carry the
                    ; payload of a response.
EOM = %xFF
OWS = <@def: " ", @match("\s*")>    ; Optional whitespace = *(SP / HTAB)
                                    ; Outgoing: single SP character
                                    ; Incoming: 0 or more SP|HTAB chars
```

The ‘Template rule’ dropdown menu will in this case only list the rules «request» and «response».

TestExpert specific prose-val commands

`<?>`

This command instructs TestExpert to present a single-line text box to the user for defining the content of the element. It is not absolutely necessary to make use of this because TestExpert will as a default always show a single-line text box when the element's content cannot be derived from a rule, i.e. when there is no definition for the rule in the ABNF file.

`<??>`

This command instructs TestExpert to present a multi-line text box to the user for defining the content of the element. This command thus allows you to overrule the default single-line text box presentation.

`<@def: out-def, in-def>`

The @def command allows you to pre-define a different value depending on whether the element is used in an outgoing context or an incoming context. A typical example where you would want to make that difference is for an element where you would want TestExpert to output for instance a single SPACE character but where TestExpert must allow zero or more space characters when incoming.

- 'out-def' must be a string (surrounded by double quotes).
- 'in-def' can be either a string (surrounded by double quotes) or the function regexpr(pattern) where 'pattern' defines the regular expression pattern for matching the character(s) that are accepted when TestExpert is validating the element for an incoming message.

Examples

`<@def: " ", regexpr(\s*)>` - With an outgoing element output a single SPACE character (0x20). With an incoming message accept zero or more whitespace characters (SPACE, HTAB).

Things you should be aware of

If you have rules that define multiple alternative elements then you have to be careful when you modify such rules; i.e. add or delete elements. When you have messages that use such a rule and you have told TestExpert to use a specific element from the rule's alternatives list TestExpert will save the selected element index in the solution file. The index is then later on used to present the correct element data to the user. That might lead to problems when you later on add or delete elements in the ABNF file. If you don't want to re-edit the message elements you should not delete alternatives and always add new alternatives at the end of the list.

Similar issues can come up when you have layed out a message definition for a Group in a solution file and the ABNF template later on comes with a modified group

definition. Depending on what type of change has occurred in the ABNF group the new/updated elements might not show correctly.
You should then temporarily change the definition mode to *Edit Mode* and then back to *Template Mode*.

Example template file

This example is for a small text protocol that can manage switches and sensors in a home control environment. There are 2 type of messages: requests and responses.

- Requests can be: “SwitchOn”, “SwitchOff”, “Status”, and “Reset”. The first 2 of those requests come with a parameter to indicate the device id (a number).
- Responses can be “OK” and “INVALID”. The “OK” response carries an additional body parameter containing a json object string. The json object contains the result of the request that has invoked the “OK” response.

Transport must be possible over TCP and so the 2 messages are terminated by means of a trailing 0xFF character.

Some message examples (the 0xFF EOM marker is not shown):

Requests:

```
SwitchOn: 2

SwitchOff: 1

Reset
```

Responses

```
INVALID

OK\r\n
{
	"id": 2,
	"name": "switch",
	"location": "living room",
	"status": "on"
}
```

This gives the following ABNF template file:

```
; !name("SimpleIOT simple-iot.abnf")
; !syntax("abnf")
; !import("core-abnf.abnf")

message = (request / response) EOM
request = switch-on / switch-off / "Status" / "Reset"
response = ok / "INVALID"
```

```
switch-on = "SwitchOn" ":" OWS device-id
switch-off = "SwitchOff" ":" OWS  device-id
device-id = <?>   ; Valid id's: 0..9
ok = ("OK" [CRLF message-body])
message-body = <??> ; The message body (if any) is used to carry the
                    ; payload of a response.
EOM = %xFF
OWS = <@def: " ", @match("\s*")>   ; Optional whitespace = *(SP / HTAB)
                                   ; Outgoing: single SP character
                                   ; Incoming: 0 or more SP|HTAB chars
```

Possible body message:

```
{
  "devices": [
    {
      "id": 1,
      "name": "switch",
      "location": "living room",
      "status": "off"
    },
    {
      "id": 2,
      "name": "switch",
      "location": "kitchen",
      "status": "on"
    }
  ],
  "status": "active"
}
```

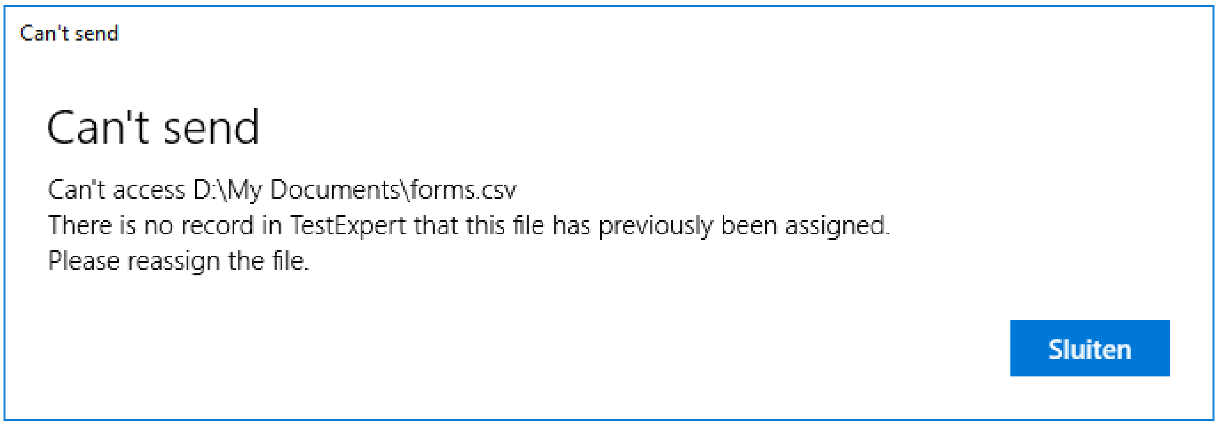
Things you should know

Including and accessing files

TestExpert, being a Universal Windows Program (UWP), only allows you to include a file (like a solution file, an ABNF template file, or a content file to be included in the body of a HTTP message) by explicitly asking you to select the file from a folder on your system.

A file added to the solution in this way is kept in a so-called Future Access List, which allows later on to access the file without the user asking again to select it.

This works perfectly as long as your solution file (which contains references to this Future Access List) is not copied to another machine and used there by another TestExpert app.
The other TestExpert app will not be able to access the files (even when you copy them over) and will show this through a suitable dialog message like the one below.


Can't send

Can't send

Can't access D:\My Documents\forms.csv
There is no record in TestExpert that this file has previously been assigned.
Please reassign the file.