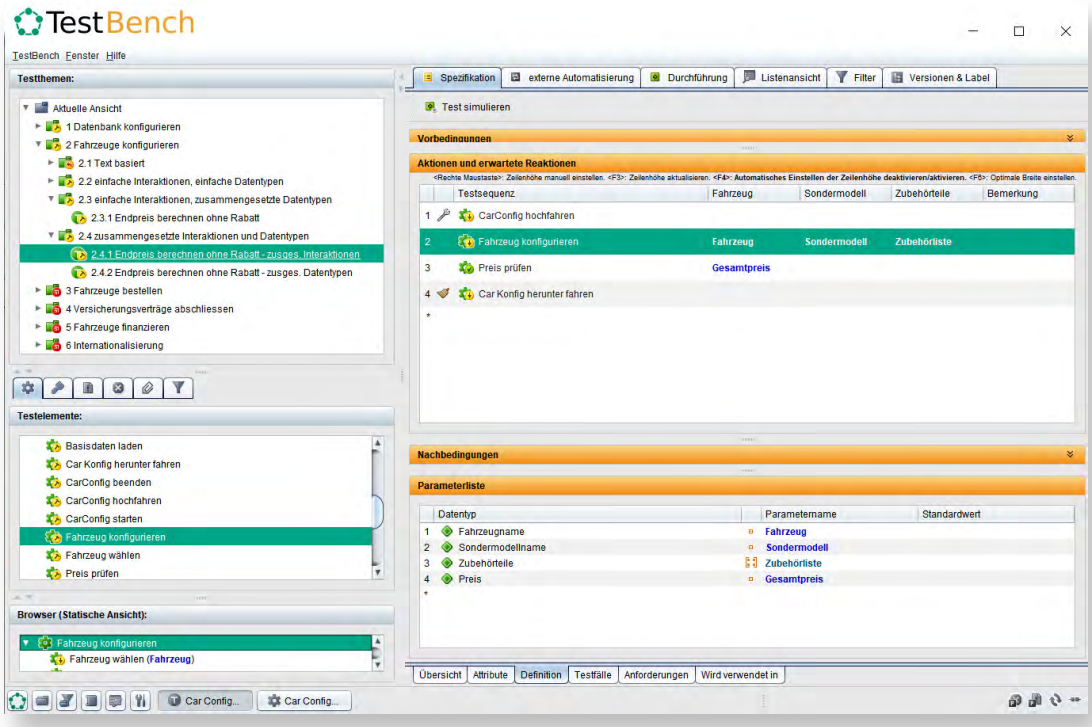
TestBench

# The Smart Testmanagement and Testdesign Tool

## Whitepaper



Whitepaper

TestBench

# No-Code Test Automation with Low-Code Integration via Robot Framework and TestBench

Author: Dierk Engelhardt
Date: 05.05.2025

imbus AG
Kleinseebacher Str. 9
91096 Möhrendorf
Deutschland
Tel. +49 9131/7518-0
Fax +49 9131/7518-50
info@testbench.com
https://www.testbench.com/

© imbus AG

## Problem Situation

In agile and continuous software development, there is an increasing need to deliver software quickly and with high quality. Test automation plays an important role in ensuring quality in an efficient and cost-effective manner. When implementing and maintaining a test automation solution, there are specific challenges that need to be overcome, which repeatedly prevent a high proportion of tests from being permanently automated:

1. Sufficient in-depth specialist knowledge of the test object and the functionalities to be tested and their interaction is required to create the right tests. At the same time, in-depth knowledge of the use of a test automation tool or framework with the appropriate development expertise is required so that the tests can be implemented and maintained in the long term.
   As a rule, QA testers have the technical knowledge of which tests are necessary and test automation specialists have the technical knowledge of how to implement these tests. These skills are rarely combined in one person.

2. If tests are to be automated, the test specification must be precise, as it is the template for the implementation. There should be no room for interpretation, as may be acceptable in manual testing. The test specification must therefore be available at the time of implementation, which is done as a subsequent step. The same applies to adaptations: The test specification must be adapted first, followed by the test automation code. Valuable time is lost in this process, which is why it is usually not possible to create the associated automated tests at the same time as the software is being developed.

## Objective: Specification = Test Automation

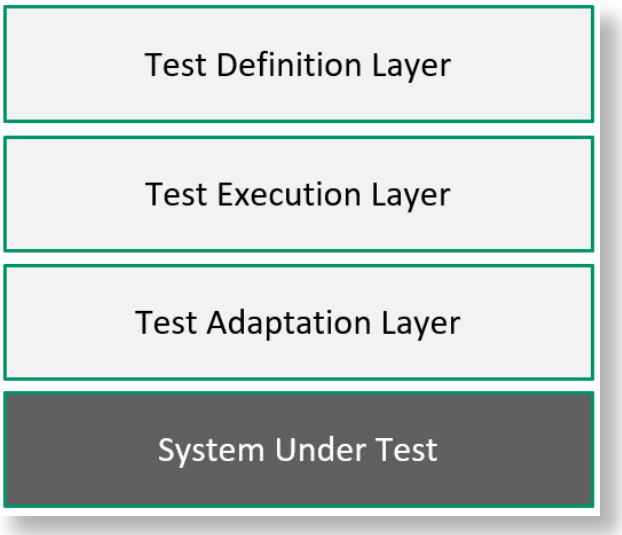
What is needed is an approach that makes it possible to specify and automate tests simultaneously without the need for QA testers to have technical expertise and test automation specialists to have domain expertise.

The best solution to these challenges is a combined no-code and low-code approach that enables both technically and professionally skilled employees to actively participate in test automation. This white paper describes a solution that combines TestBench's test design editor with the Robot Framework test framework.

## Conceptual Basis

### Generic Test Automation Architecture

The basis for the seamless interaction of the functional and technical parts of test automation is the generic test automation architecture. In it, the functional and technical parts of the solution are linked together across three levels up to the system to be tested:




Layers of the generic test automation arcitecture

The levels mean in detail:

1. **Test Definition Layer**
   The domain specific and functional test specification is created at this level. This contains the test sequences and the necessary data.
2. **Test Execution Layer**
   At this level, the tests are executed automatically, the test protocols are created and the test results are made available.
3. **Test Adaptation Layer**
   At this level, the implemented programmes or scripts suitable for the system under test are provided, which are called by the test execution layer.
4. **System Under Test**
   This layer represents the system under test with its technologies and interfaces, which are used by the Test Adaptation Layer.

© imbus AG

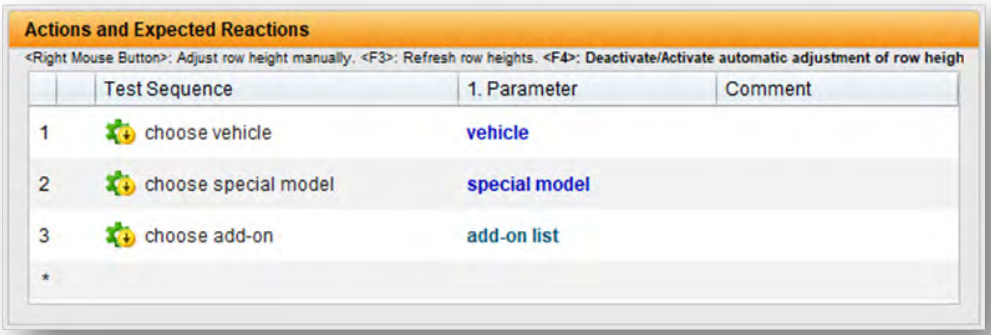
Interfaces are defined between the levels via which the levels exchange data and communicate with each other. Each level can be implemented using specialised tools.
In the case of our solution, the test design editor of TestBench represents the Test Definition Layer, the Robot Framework the Test Execution Layer and the Robot Framework Libraries the Test Adaptation Layer. TestBench and Robot Framework use the syntax of the Keyword Driven Testing as an interface between each other.

## Keyword Driven Testing with TestBench

TestBench enables test cases to be created using keyword driven testing. The integrated test design editor makes it possible to create test sequences from keywords via a user-friendly interface using drag-and-drop. The necessary test data can also be defined and used as parameters or data tables. Keywords and test data are managed in libraries that enable quick and easy access.

A keyword can in turn be composed of keywords so that detailed processes can be mapped as more abstract, easier to understand processes through aggregation. This creates tests in understandable language that also consider the necessary level of detail. The use of keywords creates clear test points, as a test result is later determined for each keyword during the test execution. The use of keyword driven testing also means a high degree of reuse, as keywords defined once are called up again and again. If changes are made, only individual keywords need to be adapted, which also significantly reduces the maintenance and servicing effort.
An example illustrates the advantages and possibilities of this method:

A vehicle consisting of a basic model, special model and several add-ons is to be configured. The test sequence required for this is mapped using three keywords:



The parameter list of the test sequence maps the parameter interface of the test sequence:

**List of Parameters**

| | Data Type | Parameter Name | Default Value |
|---|---|---|---|
| 1 | vehicle name | vehicle | |
| 2 | special model name | special model | |
| 3 | add-on name | add-on list | |
| * | | | |

The values of the test sequence parameters are stored in data types and can be assigned values in the associated data table:

**Test Cases**

| | vehicle | special model | add-on list |
|---|---|---|---|
| 1 | *Rolo* | - | [-] |
| 2 | *Rassant Family* | *Jazz* | [RSF] |
| 3 | *I5* | *Luxus* | [all] |
| * | | | |

Each line of this table represents a run of the test sequence with the respective values and thus represents a specific test case.

This test sequence with its three keywords can now be mapped in the same way in a new keyword that abstracts the vehicle configuration. This new keyword can in turn be inserted into a new test sequence for ordering a vehicle:

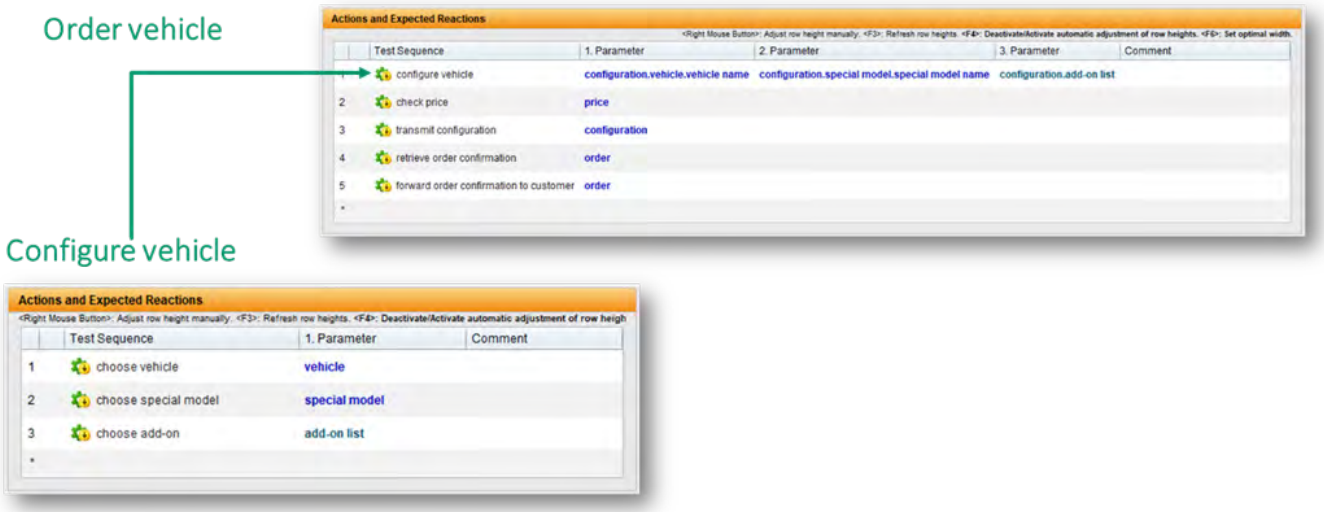
**Actions and Expected Reactions**

<Right Mouse Button>: Adjust row height manually. <F3>: Refresh row heights. <F4>: Deactivate/Activate automatic adjustment of row heights. <F6>: Set optimal width.

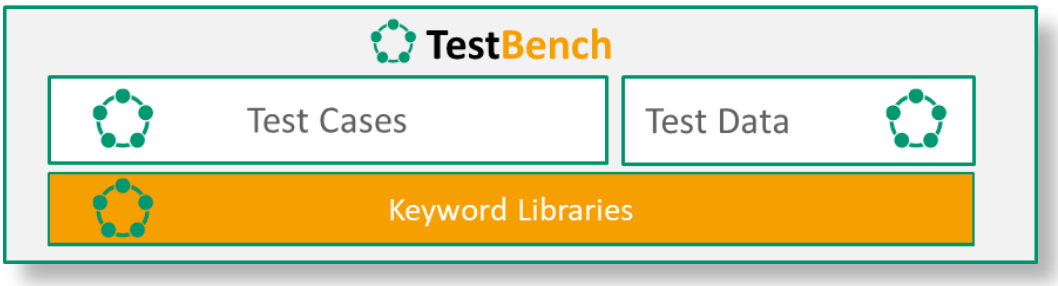
| | Test Sequence | 1. Parameter | 2. Parameter | 3. Parameter | Comment |
|---|---|---|---|---|---|
| 1 | configure vehicle | configuration.vehicle.vehicle name | configuration.special model.special model name | configuration.add-on list | |
| 2 | check price | price | | | |
| 3 | transmit configuration | configuration | | | |
| 4 | retrieve order confirmation | order | | | |
| 5 | forward order confirmation to customer | order | | | |
| * | | | | | |

Like the keywords, the three parameters can be organised into structures. In the illustration above, the example shows how the structure is used to map a complete vehicle configuration, which is transferred to subsequent keywords after a single configuration.

The following image emerges of a very comprehensible but sufficiently detailed test specification using keyword driven and data driven testing:




The test definition layer is therefore as follows:




**TestBench thus represents the No-Code part oft he solution.**

## Keyword-Driven Testing with Robot Framework

Like TestBench, the Robot Framework is based on a keyword driven testing approach in which individual test actions are also defined by keywords. It makes it possible to write tests with an easy-to-read, tabular syntax that is easy to understand even for non-technicians. Basic programming knowledge is an advantage when creating tests.
Robot Framework already offers built-in standard libraries (e.g. BuiltIn, OperatingSystem, Collections) and can be extended by external libraries. Several hundred keyword libraries are available that contain ready-made implementations for technologies to be tested, such as the Selenium Library or the Browser Library for testing web interfaces or the Robosapiens Library for testing SAP. Technical actions such as UI interactions, API queries or database operations can thus be executed using simple commands that have already been defined. Only when these libraries are no longer sufficient is it necessary to programme your own libraries or additions to libraries, and only then is in-depth programming knowledge required.
Development for Robot Framework is usually carried out using development environments such as Microsoft Visual Studio Code or JetBrains IntelliJ Idea. These development environments are usually extended by extensions, such as Robotcode for VS Code, so that functionalities such as code completion are also available for Robot Framework syntax. This makes these environments fully-fledged development tools for Robot Framework, which are in no way inferior to development environments for programming languages such as C# or Python.

A test case for configuring a vehicle could look like this in Robot Framework (here using VS Code and the RobotCode extension):

```
Configure vehicle
    Select Base Model     Rolo
    Select Special Model     Luxus
    Select Accessory     Alloy Sport Rims
```

The test sequences are easy to read in the tabular representation and the parameters can be quickly assigned. A keyword such as ‘Select Base Model’ is followed by an implementation specific to the system under test, which is carried out by test automation engineers:

```
Select Base Model
    [Arguments]    ${basemodel}
    Click    css=[href="/config/basemodel"]
    Wait For Elements State    ${select_CarBaseModel}
    Select Options By    ${select_CarBaseModel}    text    ${basemodel}
```

At this level, ready-made keywords (e.g. ‘Click’ or ‘Select Options By’) from the Open Source Browser Library for Robot Framework are used in our example, so that the Robot Framework syntax is used up to this step.

Another very important part of Robot Framework is its ability to determine the results of automated test execution, handle exceptions and present them in very informative logs. Clear HTML reports are generated for each execution, containing statistics on success, failure, duration, etc.

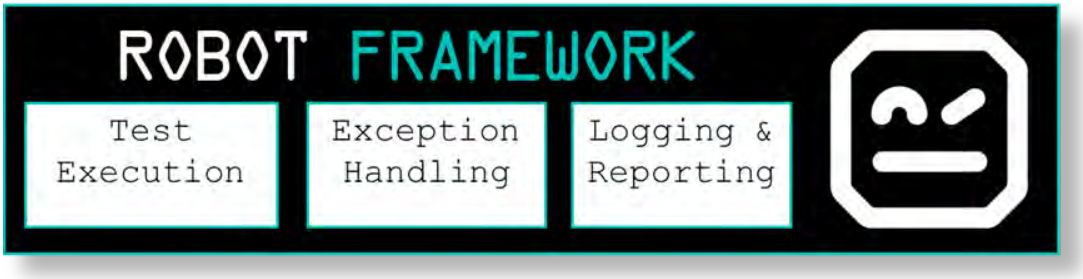
A log of our example looks like this after a successful execution:

```
GROUP  Select Vehicle                                                          00:00:01.646
Start / End / Elapsed:   20250420 11:24:56.562 / 20250420 11:24:58.208 / 00:00:01.646
KEYWORD  functional_keywords.Select Base Model  Rolo                           00:00:01.596
Start / End / Elapsed:   20250420 11:24:56.569 / 20250420 11:24:58.165 / 00:00:01.596
KEYWORD  Browser.Click  css=[href="/config/basemodel"]                         00:00:00.087
Documentation:   Simulates mouse click on the element found by selector.
Tags:            PageContent, Setter
Start / End / Elapsed:   20250420 11:24:56.579 / 20250420 11:24:56.666 / 00:00:00.087
11:24:56.588  INFO  Clicks the element 'css=[href="/config/basemodel"]'.
KEYWORD  Browser.Wait For Elements State  ${select_CarBaseModel}               00:00:01.368
Documentation:   Waits for the element found by selector to satisfy state option.
Tags:            PageContent, Wait
Start / End / Elapsed:   20250420 11:24:56.676 / 20250420 11:24:58.044 / 00:00:01.368
11:24:58.043  INFO  Waited for Element with selector "Basismodell" >> ../.. >> select at state visible
KEYWORD  Browser.Select Options By  ${select_CarBaseModel}  text  ${basemodel}  00:00:00.108
Documentation:   Selects options from select element found by selector.
Tags:            PageContent, Setter
Start / End / Elapsed:   20250420 11:24:58.048 / 20250420 11:24:58.156 / 00:00:00.108
11:24:58.056  INFO  Selects the option(s) Rolo by attribute SelectAttribute.label from element "Basismodell" >> ../.. >>
                    select.
```

For each step of the test procedure, the results can be viewed down to the lowest level of keywords.

In the event of an error, the problem can also be narrowed down to these lowest levels, which is a decisive advantage when analysing the cause.

The test execution layer is therefore as follows:




**Robot Framework thus represents the Low-code part of our solution.**

The implementation of the keyword ‘Click’ in our example in the Browser Library now looks as follows:

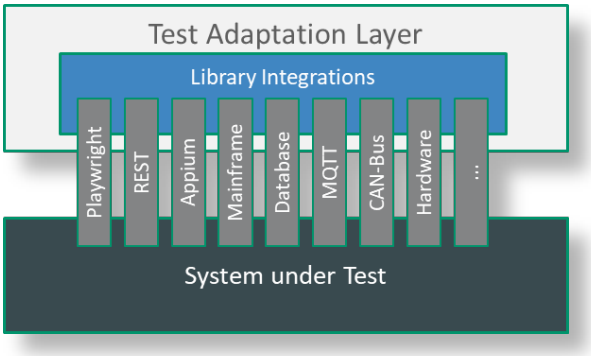
```
@keyword(tags=("Setter", "PageContent"))
def click(self, selector: str, button: MouseButton = MouseButton.left):
    """Simulates mouse click on the element found by ``selector``.

    This keyword clicks an element matching ``selector`` by performing the following steps:
    - Find an element matches selector. If there is none, wait until a matching element is attached to the DOM.
    - Wait for actionability checks on the matched element, unless ``force`` option is set.
      If the element is detached during the checks, the whole action is retried.
    - Scroll the element into view if needed.
    - Use `Mouse Button` to click in the center of the element, or the specified position.
    - Wait for initiated navigation to either succeed or fail.


    | =Arguments= | =Description= |
    | ``selector`` | Selector element to click. See the `Finding elements` section for details about the selectors. |
    | ``button`` | Defaults to ``left`` if invalid. |

    Keyword uses strict mode, see `Finding elements` for more details about strict mode.

    Example:
    | `Click`    id=button_location
    | `Click`    id=button_location    left
    | `Click`    id=button_location    right

    [https://forum.robotframework.org/t//4238|Comment >>]
    """
    self.click_with_options(selector, button)
```

The Browser Library provides a ready-made library for Robot Framework for testing web interfaces based on Playwright. Playwright, in turn, is an open source test automation framework developed by Microsoft. It enables tests for web applications across different browsers. Playwright supports the automation of Chromium (e.g. Chrome, Edge), Firefox and WebKit (e.g. Safari) with a single, consistent API. All keywords are very well documented in the library, so that they can be easily used in your own tests with basic programming knowledge. If these keywords are to be expanded, in-depth programming knowledge is required from this point onwards.

The call structure is similar to the structure in TestBench, but does not cover the functional part of the tests, but maps the functional tests to the technical level of the system under test:

```
[Robot Framework Test Cases]
    |
    |→ [Keywords of the System under Test]
            |
            |→ [Robot Framework Libraries]
                    |
                    |→ Browser Library
```

Thanks to the possibility of integrating many ready-made libraries and the option of adapting these to the specific characteristics of the system under test or even creating your own libraries, almost any technical problem can be solved.
At this lower level, it is also very easy to address different technologies within a test case, as different libraries for various technologies can be used simultaneously by mapping the functionalities to the technology. End-to-end tests, starting in the browser, via SAP and mobile devices back to the browser, suddenly become very easy to realise.
This means that the Test Adaptation Layer is also included in our solution:

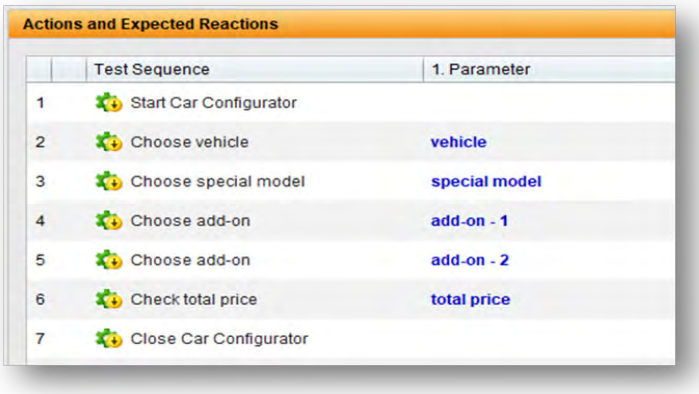


## Test Design meets Automation

The core idea of the solution is that keywords from the Robot Framework are made available directly in TestBench and abstracted there. Test designers can thus create test scenarios at a higher level, while technical experts define and maintain specific technical keywords. In addition to this bottom-up approach, it is of course also possible to take a top-down approach and create the functional parts first, which are then realised at the technical level.
In reality, QA testers and test automation experts will work simultaneously and in parallel, which is exactly what the approach of integrating TestBench with Robot Framework pursues. There is no single or multiple import or export to the other system; instead, the systems are continuously synchronised with each other. TestBench serves as the leading system that generates test cases and assumes that keywords have already been technically implemented. The QA testers create new keywords that the test automation engineers can implement immediately, just as the QA testers immediately recognise that new keywords have been implemented that they can use in their tests.
A TestBench extension for Robot Framework exists to optimise this smooth collaboration, which exchanges all the necessary information between the two tools via the TestBench API. This means that both groups can remain in their respective domains, do not have to change tools and are still always up to date. This ensures a high degree of parallelism when creating automated tests and an equally high reaction speed on both sides in the case of innovations and changes.

TestBench is also the leading system for the execution of test cases. If test cases are to be executed, the system generates the test cases for Robot Framework and assumes that the keywords used in them are implemented in Robot Framework.

The test sequence from TestBench is thus mapped 1:1, which means that it is not necessary for the test automation specialists to have domain specific expertise.



Actions and Expected Reactions

| | Test Sequence | 1. Parameter |
|---|---|---|
| 1 | Start Car Configurator | |
| 2 | Choose vehicle | vehicle |
| 3 | Choose special model | special model |
| 4 | Choose add-on | add-on - 1 |
| 5 | Choose add-on | add-on - 2 |
| 6 | Check total price | total price |
| 7 | Close Car Configurator | |

The associated test data is defined in TestBench as follows:

Test Cases

| | UID | vehicle | special model | add-on - 1 | add-on - 2 | total price |
|---|---|---|---|---|---|---|
| 1 | TB-TC-161-PC-442 | Rolo | Luxus | alloy sport rims | leather steering wheel | 16,059.99 |

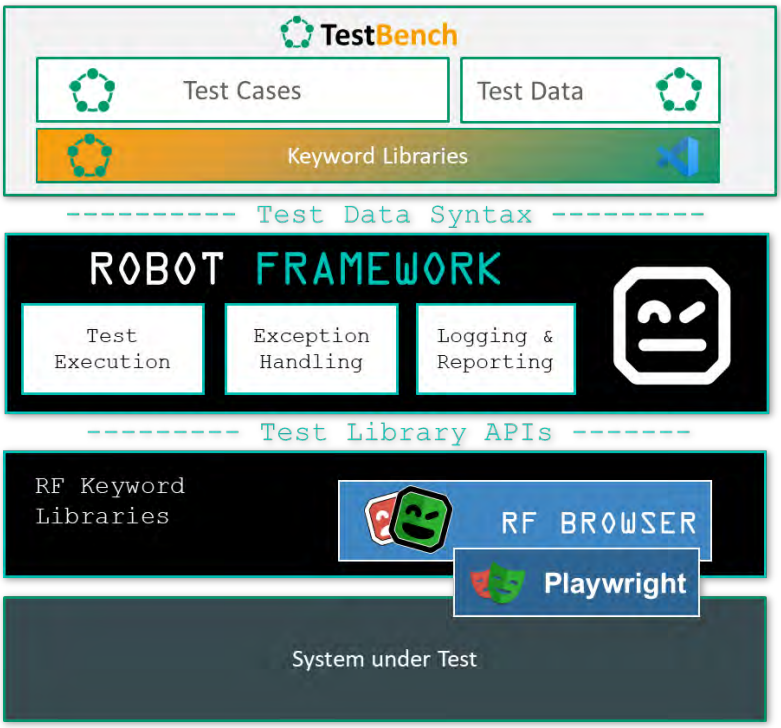
The test sequence and the data set of the test case from TestBench are generated as a Robot Framework test case:

```
iTB-TC-337-PC-1659
    GROUP    CarConfig starten
        Open CarConfig
        Set Username    schulung20
        Set Password    @RBTFRMWRK@
        Click Login Btn
    END
    GROUP    Fahrzeug wählen
        Select Base Model    Rolo
    END
    GROUP    Sondermodell wählen
        Select Special Model    Luxus
    END
    GROUP    Zubehör wählen
        Select Accessory    Sportfelgen
    END
    GROUP    Zubehör wählen
        Select Accessory    Lederlenkrad
    END
    GROUP    Endpreis bitte prüfen
        Verify Total Price    16,059.99    €
    END
    GROUP    CarConfig beenden
        Close CarConfig
    END
```

The above example shows that there is another level in TestBench below the functional level (recognisable by the GROUP elements), the so-called navigation level. In our case, it maps the functional level to the operating sequence in the system under test. This navigation level represents the transition to the Robot Framework. It could also be mapped in the Robot Framework, but then the test automation engineers must have basic knowledge of the system under test. The level at which the transition takes place can be determined on a case-by-case basis, but the procedure from the example has proven itself many times over.

In addition, the test data are used as transfer parameters for the individual keywords as constants when the tests are generated. This means that no administration for test data needs to be set up in Robot Framework, as this is also taken from TestBench. Test automation engineers do not need to have any technical knowledge of the test data either, only knowledge of the type of data and the way in which it must be transferred to the system under test or read from there.

The combination of TestBench and Robot Framework maps all levels of the generic test automation architecture:




**This achieves the goal: specification = Test Automation!**

## Advantages of a Multi-Level No-Code/ Low-Code Solution

The combination of no-code and low-code tools into an integrated solution offers many advantages:

- **Simplified test design**: Business users can create test cases independently without in-depth technical knowledge.
- **Centralised keyword library**: Domain specific and technical keywords are managed in central, clearly organised keyword libraries. Each level in the tool best suited to it.
- **Improved collaboration**: Clear separation and yet close integration between technical implementation and domain specific test design.
- **Increased efficiency and maintainability**: Easier adjustments thanks to centralised management of keywords and simple maintenance of test cases.
- **Parallel working**: Domain specific tests can be created at the same time as the technical realisation and are merged in the respective tools.

- **Interchangeability of the levels**: The levels of the test automation architecture can each be replaced, modernised or extended individually. This is particularly relevant for the Test Adaptation Layer, as technological progress is the most volatile and fastest.

The solution can optimally meet the two challenges mentioned at the beginning:

1. People with domain specific knowledge and people with technical knowledge can work together smoothly and create test automation together very quickly and efficiently.
2. Parallel work by the experts enables rapid progress so that the realisation of automated tests can keep pace with the development of the functionalities to be tested.

Compared to highly specialised no-code test automation tools, the solution presented is much more flexible, can be modified more easily at all levels and uses a type of presentation and documentation that can be used very well for small numbers of tests, but also for large test suites.

© imbus AG

# Contact


**imbus AG**

Kleinseebacher Str. 9
91096 Möhrendorf
DEUTSCHLAND

Tel. +49 9131 7518-0

info@testbench.com

www.testbench.com