RANOREX

STUDIO ADVANCED USERGUIDE

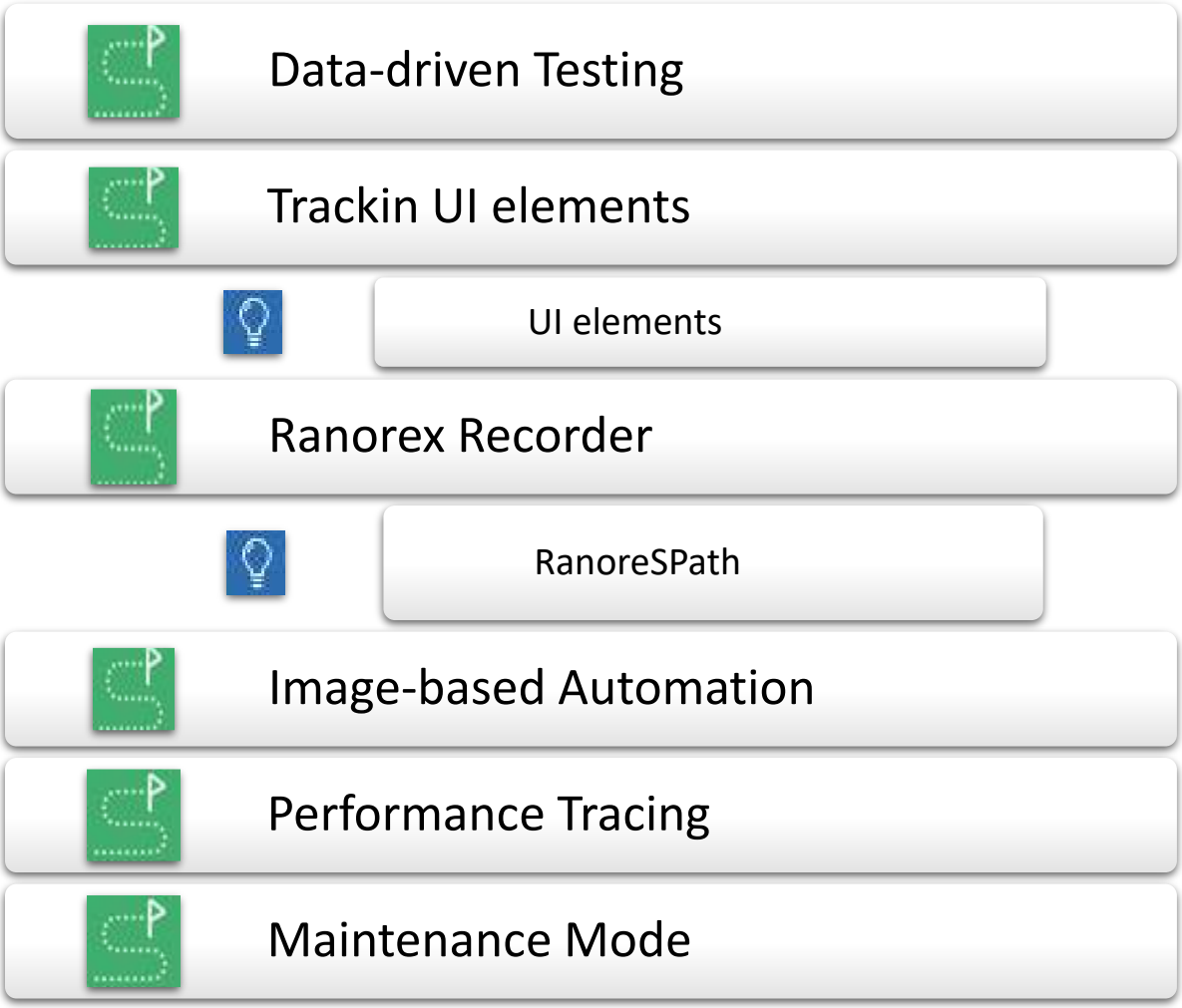
TABLE OF CONTENTS

RANOREX STUDIO ADVANCED 2

DATA-DRIVEN TESTING 3
Define test and variables 5
Manage and assign data sources 18
Data binding 36
Run a data-driven test 45
Parameters 50
Conditions and rules 61
TRACKING UI ELEMENTS 68
Track by recording 69
Track button 72
Instant tracking 75
RANOREX SPY 77
Browser & results screen 82
The path editor 90
Snapshot files 99
GDI capture feature 106
UI ELEMENTS 113
UI elements 113
Roles, capabilities, and more 119
RANOREXPATH 127
RanoreXPath basics 128
RanoreXPath examples 135
RanoreXPath Syntax 155
IMAGE-BASED TESTING 163
Image-based testing basics 167
Advanced image-based testing 170
Image editor 178
Image-based properties 180
MAINTENANCE MODE 185
PERFORMANCE TRACING 190
Trace logs 191
Leverage the data 196

Ranorex Studio advanced

This section addresses advanced topics concerning automated software testing with Ranorex Studio. Here you will find detailed information on test solution processes and methods, tools, and advanced concepts.



Meaning of symbols

Method: Contains the detailed description of a process. This includes an explanation of the process, the goal(s) of the process, and a step-by-step explanation of the actions necessary to successfully complete the process

Concept: Detailed description of a principle, technical concept or key idea applied within one or more Ranorex Studio tools and/or methods. These concepts are important for understanding and using the referenced tool or method.

Data-driven testing

In data-driven testing, a test container (test case/smart folder) retrieves input values from a data source such as an Excel spreadsheet or a database file. The test container is then repeated automatically for each row of data in the data source.

The key components of a data-driven test are variables, data sources, and data binding. You'll learn how to combine these to build a data-driven test in the following chapters. A sample solution and accompanying instructions will guide you through the process.

Parameters are another component of data-driven testing. They increase module reusability and make it possible to pass variable values from one recording module to another.

Data-driven testing also allows you to use conditions to further control test execution, which is why this topic is explained at the end of the data-driven testing chapter.

Screencast

The screencast "What is data driven testing?" walks you through information found in this chapter.:

Watch the screencast now

Download the sample solution

Except for the chapter on Conditions, each chapter in this section is dedicated to a separate part of building a data-driven test in Ranorex Studio and contains the necessary step-by-step instructions. Use the sample solution available below to put these instructions into practice. It comes with a prepared test suite and the required modules.

There is also a completed sample solution available in the chapter ⇢ Run a data-driven test. It contains the finished project after all the steps in the chapters have been completed. You can use it as a reference.

Sample Data Driven Testing

Install the sample solution:

1. **Unzip** to any folder on your computer.

2. **Start** Ranorex Studio and **open** the solution file `RxSampleDataDrivenTesting.rxsln`

> **Hint**
>
> The sample solution is available for Ranorex versions 8.0 or higher. You must agree to the automatic solution upgrade for versions 8.2 and higher.

The process of data-driven testing

To get you started, here's a quick overview of how data-driven testing works:

A **test case** contains the **recording modules** that make up the test. The recording modules, in turn, contain the **actions** that are carried out during the test run and the **repository items** these actions are performed on.

These actions and repository items can be made **variable** and **bound** to an **external data source** which provides the **test data**. During the test run, these variables are then **fed with the values** from the data source. The test is **driven by data**; therefore, it's a "data-driven" test.

Like other actions, you can make **validations variable** as well. This allows you to **verify** whether the test data entered during a data-driven test produces the correct results in the AUT, i.e. if the actual result corresponds to the expected result. This is optional, but very useful. The sample solution that we'll build in the following chapters includes a test-driven validation.

Recommended knowledge

To get the most from the examples in this chapter, we recommend a basic understanding of the material below:

Reference

To follow this chapter, you should have a working knowledge of the Ranorex fundamentals chapter, with the following chapters in particular:

- ⇢ Ranorex Recorder
- ⇢ Test suite
- ⇢ Actions
- ⇢ Repository
- ⇢ Test validation

Define test and variables

In this chapter, we'll first define the steps of the data-driven test. Then, we'll define the required variables for this scenario. You'll also find out how to manage variables in Ranorex Studio.

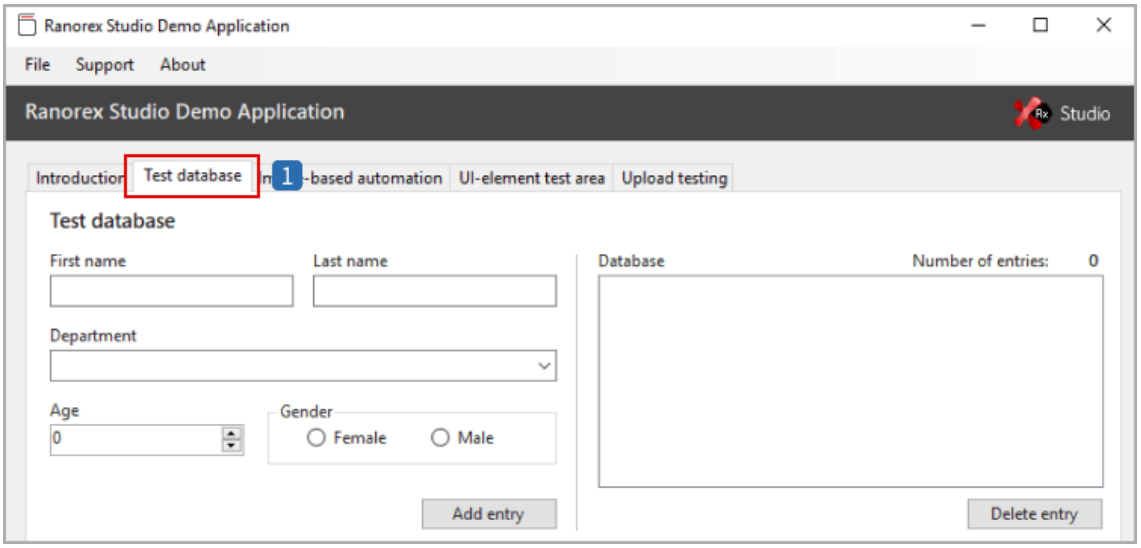
Variables are one of the key parts of data-driven testing. They are the placeholders for values you want to feed your test, either from data sources or parameters. In Ranorex Studio, we differentiate between three types of variables: Action variables, their subtype validation variables, and repository variables.

Test definition

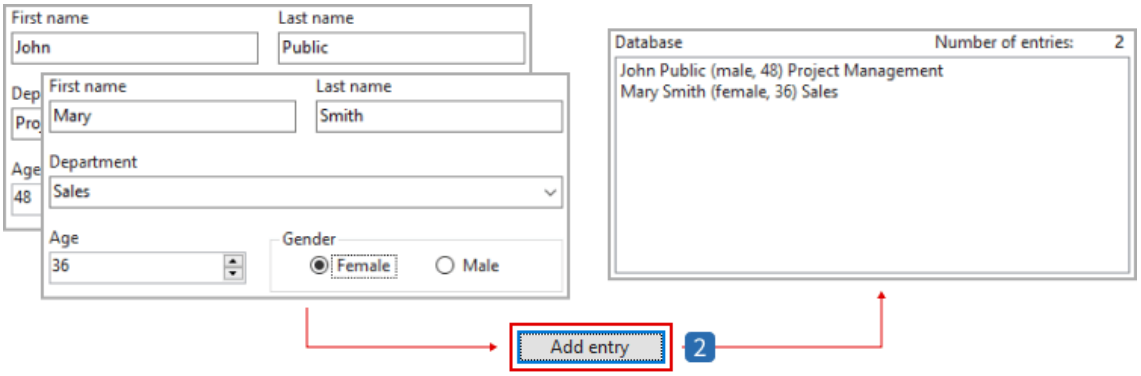
As usual, we'll use the Ranorex Studio Demo Application for our example. It has a database function that we'll test. Testing a database is a common real-world application of a data-driven test.

The test will contain the following steps:

1. **Start** the Demo App.
2. **Click** the **Test database** tab.

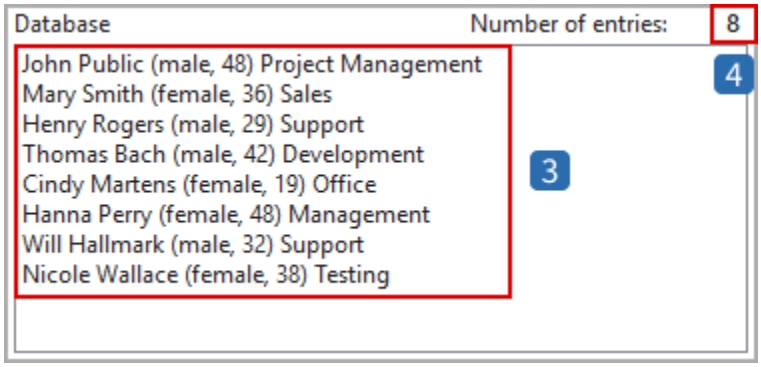


3 **Add** an entry to the database.



3 **Repeat** until there are 8 entries in the database…

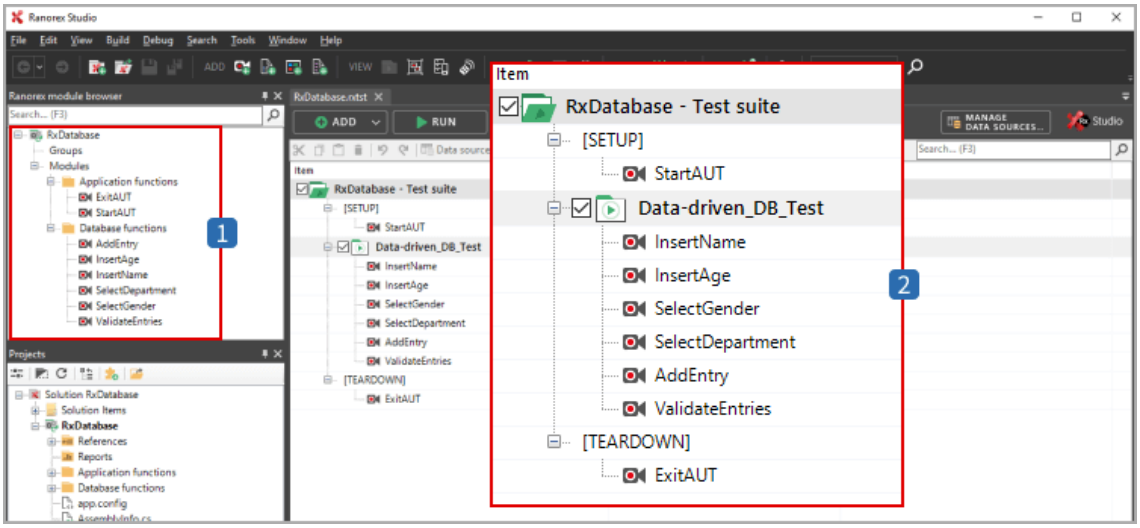
4 …and after each entry, **validate** that the **Number of entries** updates correctly.



5 **Close** the Demo App.

Sample solution and prepared modules

The ⇢ sample solution already contains all the modules with the necessary actions for the steps above. They are organized in a simple test suite. This will serve as our starting point for the next step: replacing constant values in several modules with variables.



1 **Module browser** with recording modules structured in folders

2 **Test suite** with a ready-to-run database test scenario

Define action variables

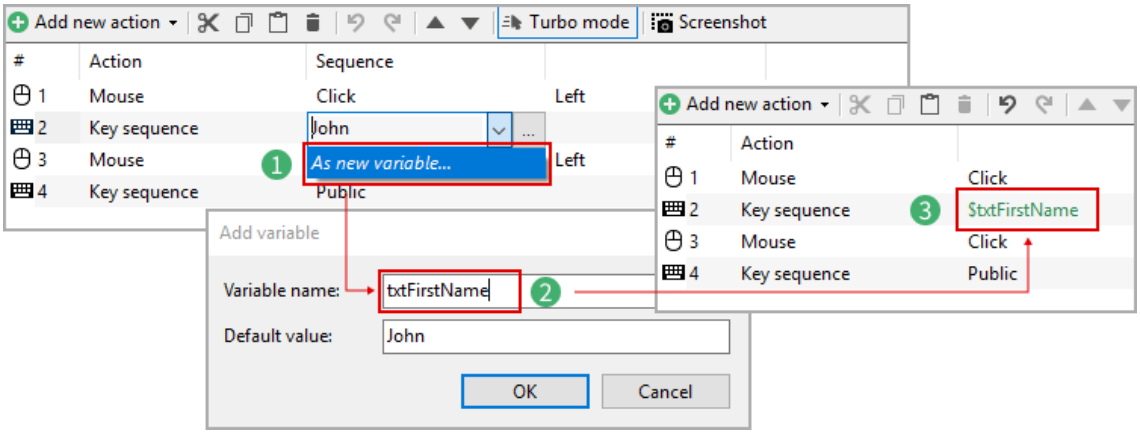
Action variables **replace a constant value in a component or property of an action**, e.g. the specific text string in a Key sequence action. Action variables are **limited to the recording module they're defined in**.

First, we'll replace the constant text values in the respective actions for entering the first name, last name, and age of a person in the Demo App database.

> **Note**
>
> In Ranorex Studio, variable names have the pattern `$<variablename>`, where <variable name> cannot start with a number or contain any special characters, except for underscores.

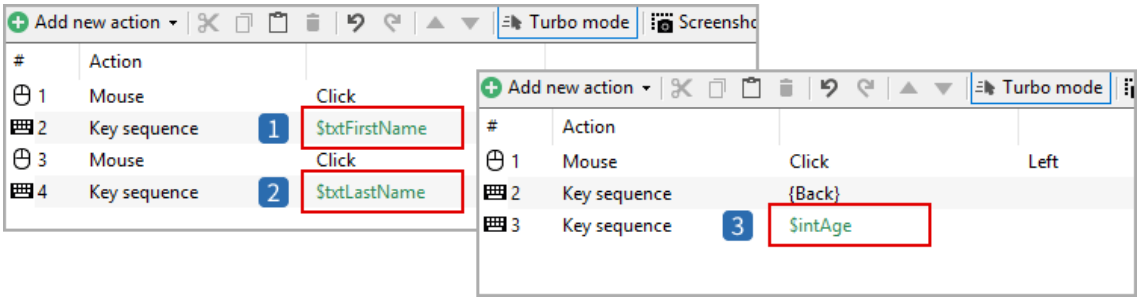
To replace a constant value with a variable:

1. **Open** the drop-down list for the constant value you want to change and **click As new variable...**
2. **Name** the variable so you can easily identify what value it is a placeholder for and **click OK.**
3. The variable appears in the action table in **green**, replacing the constant value.



4. Repeat the above process for the last name (InsertName module) and the age (InsertAge module).

The final result should look like this:



1. Variable `$txtFirstName` replacing the constant value ‘John’ for the first name
2. Variable `$txtLastName` replacing the constant value ‘Public’ for the last name
3. Variable `$intAge` replacing the constant value ’48’ for age

> **Note**
>
> Action variables can only be used in the recording module you defined them in.

Define repository variables

Aside from making action values variable, data-driven testing often involves creating repository variables. Repository variables **replace a certain part of an item’s RanoreXPath, the defining attribute,** with a variable. You can make any repository item variable this way. Repository variables are **available in the repository they were defined in and all modules that reference this repository**.

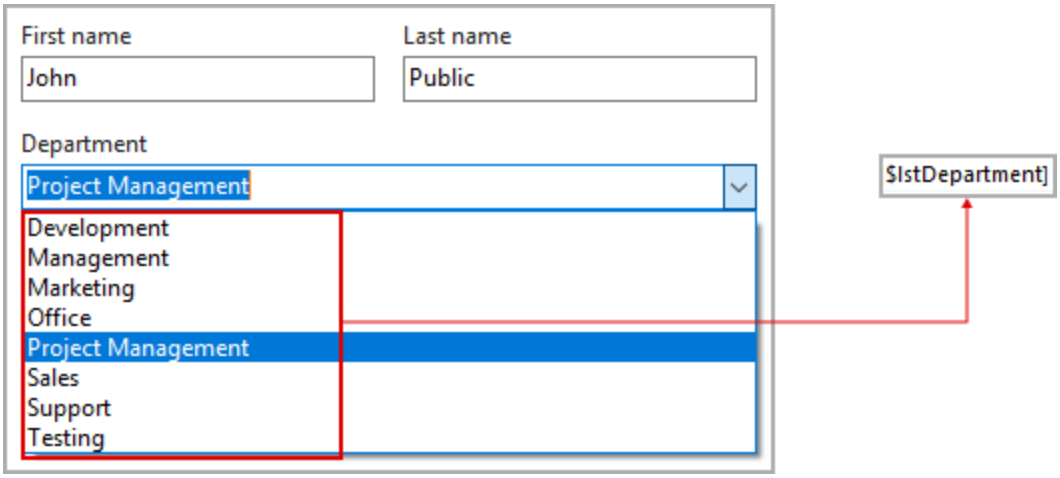
For our example, we’ll make a list selection and radio buttons variable. These are repository items where it often makes sense to make them variable because they are usually part of a selection process with multiple options.

> **Reference**
>
> For more information on RanoreXPath, go to Ranorex Studio advanced > ⇢ RanoreXPath

Make a list selection variable

Here, we’ll make the repository item for the Department list selection variable. In the module **SelectDepartment** of the sample solution, the **Click** action is linked to a constant repository item that points to the **Project Management** entry in the list. In the item’s RanoreXPath, the attribute **@text=’Project Management’** is responsible for this.

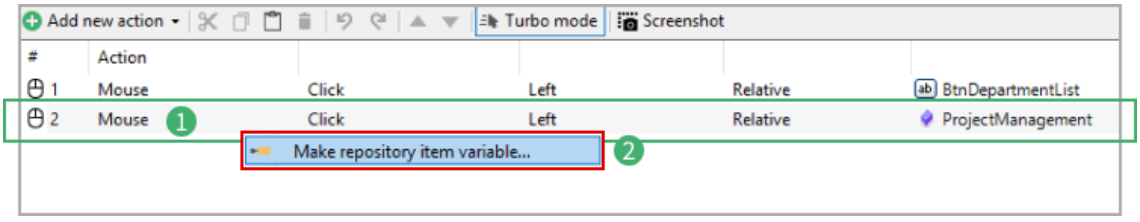
We will now replace the constant attribute value **Project Management** with the variable **$lstDepartment** so the Click action is performed on whatever list entry our future data source specifies.



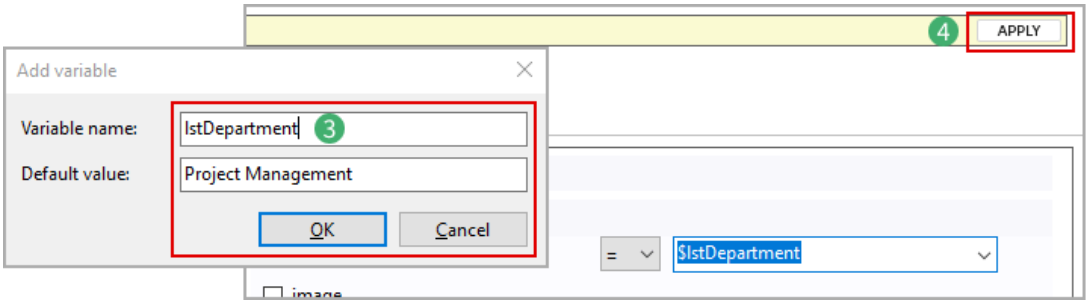
> **Note**
>
> When we create our data source later, we will need to make sure the entries for the department selection are exactly the same as in the list, or else Ranorex Studio won't be able to identify the UI elements. For example, there is no entry **IT**, so a repository item with the resulting attribute **@text='IT'** wouldn't point to anything.

To define the variable:

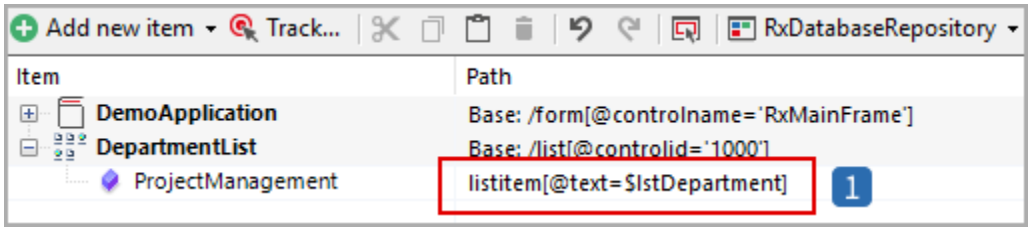
1. **Right-click** the second **Click** action and choose **Make repository item variable…**



2. Ranorex Spy opens with the constant value of the text attribute already highlighted. **Click** the variable symbol next to it.
3. **Name** the variable so you can easily identify what repository item it refers to and **click OK**.
4. In the yellow bar, **click Apply.**



5 In the repository, you can now see the variable in the item's RanoreXPath.

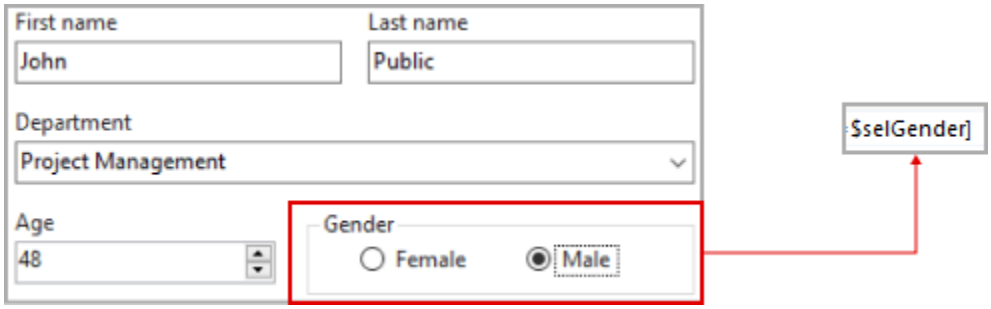


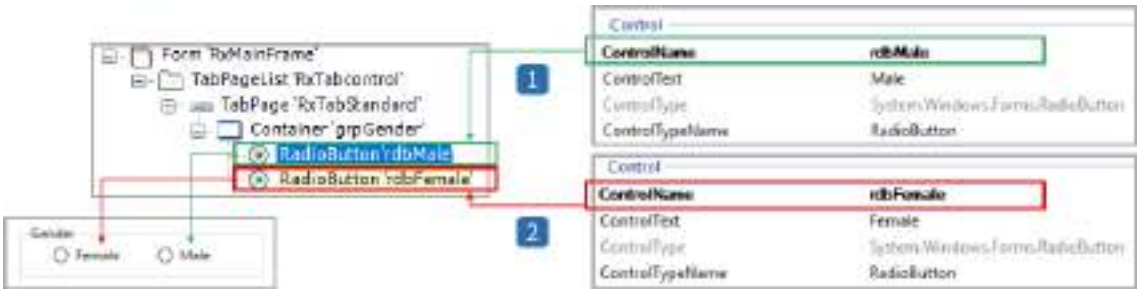
1 Repository variable `$lstDepartment` replaces the constant value for department selection.

Make radio buttons variable

Here, we'll make the repository item for the **Gender** selection variable. In the module **SelectGender** of the sample solution, the **Click** action is linked to a constant repository item that points to the radio button for **Male**. In the item's RanoreXPath, the attribute **@controlname='rdbMale'** is responsible for this.

We will now replace the constant attribute value in the RanoreXPath of this repository item with a variable (**$selGender**) so the Click action is performed on whatever Gender radio button our future data source specifies.





Note

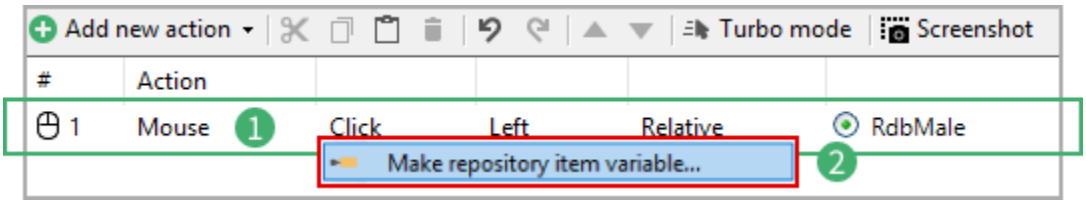
When we create our data source later, we will need to make sure the entries for the gender selection are exactly the same as the respective @controlname attributes of the two radio buttons, i.e. **rdbMale** and **rdbFemale**, or else Ranorex Studio won't be able to identify the UI elements. A data source with just Male and Female would not work, because the repository item with the resulting attribute **@controlname='Male'** wouldn't point to anything.

Hint

To make a data-driven testing robust against localization (other languages), it is recommended to use attributes such as `ControlName`.

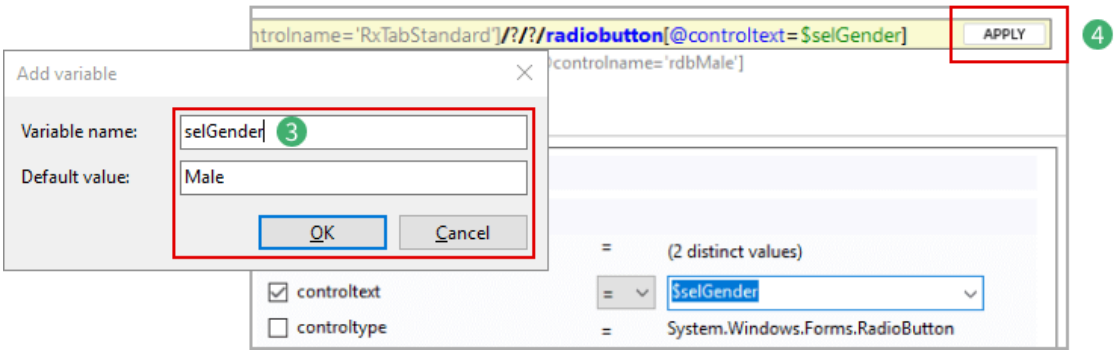
To define the variable:

1. **Right-click** the **Click** action and choose **Make repository item variable...**

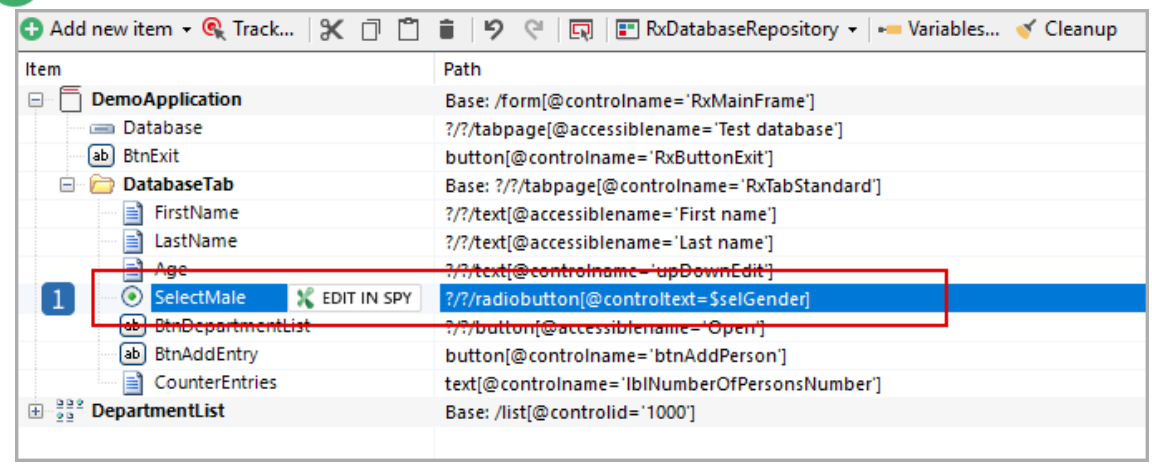


2. Ranorex Spy opens with the constant value of the text attribute already highlighted. **Click** the variable symbol next to it.

3. **Name** the variable so you can easily identify the repository item it refers to and **click OK**.
4. In the yellow bar, **click Apply.**



5. In the repository, you can now see the variable in the item's RanoreXPath.



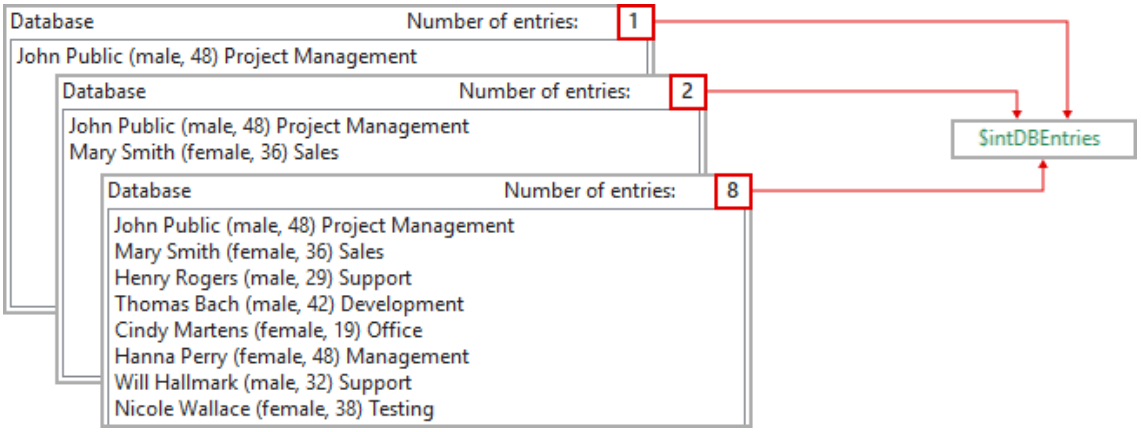
1. Repository item representing the **Gender** radio button selection with variable `$selGender`

> **Note**
>
> Repository variables can be used in the repository you defined them in and all recording modules that use this repository.

Define validation variable

Our example test includes a step that validates whether the database counter updates correctly as entries are added. For this validation to work, we need to make it variable.

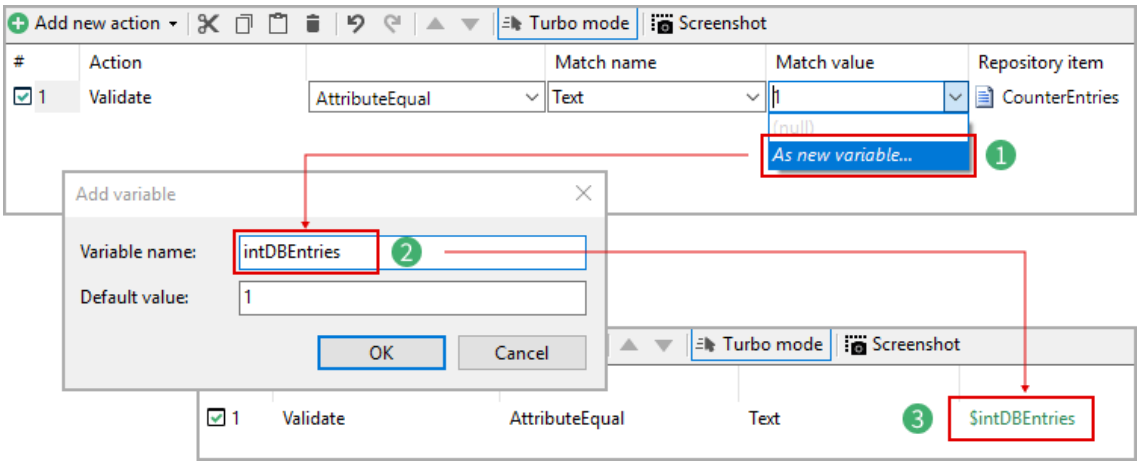
So, we'll replace the constant Match value of the validation action in the ValidateEntries module with a variable.



> **Note**
>
> In principle, validation variables are the same as action variables. However, as the Validation action is more complex than most others, we treat their variables separately and therefore also explain the process separately.

To define the variable:

1. **Open** the drop-down list for the Match value and **click As new variable…**
2. **Name** the variable so you can easily identify the value it is a placeholder for and **click OK.**
3. The variable appears in the action table in **green**, replacing the constant value.



Default values

The default value of a variable is the value that's used when no data source is available. This is the case when running a recording module from the recording module view, for example. Therefore, you should always make sure to define a meaningful default value.

> **Note**
>
> When you replace a constant value with a variable, the original constant value is automatically used as the default value.

View and manage variables

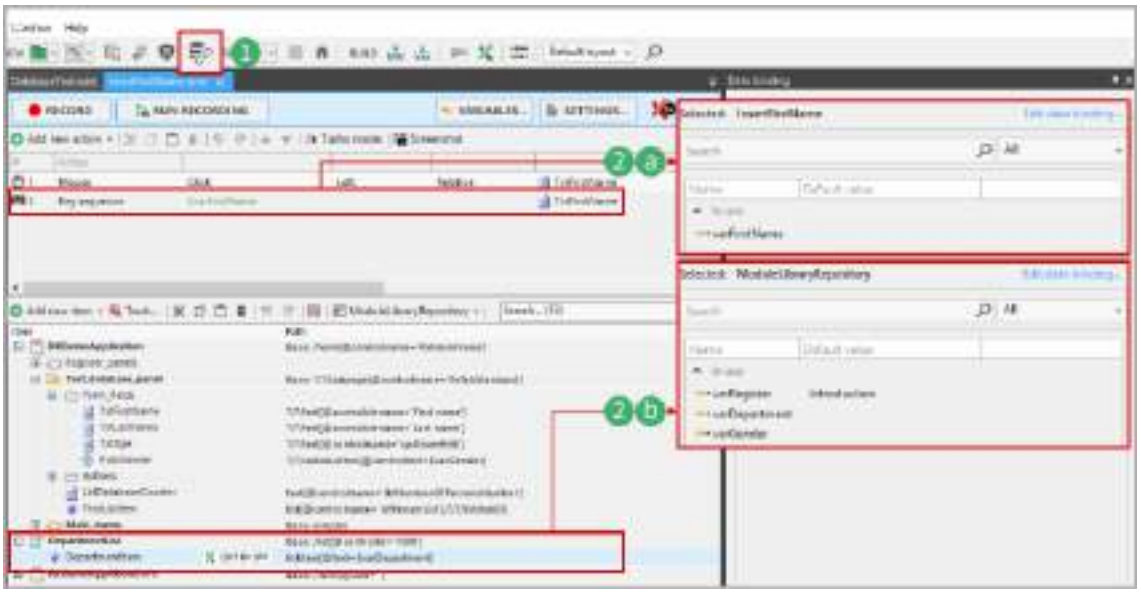
You can view the status of variables and manage them in different places: the data binding pad gives you a detailed overview of the status of all variables in a recording module, while the VARIABLES… dialog allows you to add variables and to edit them.

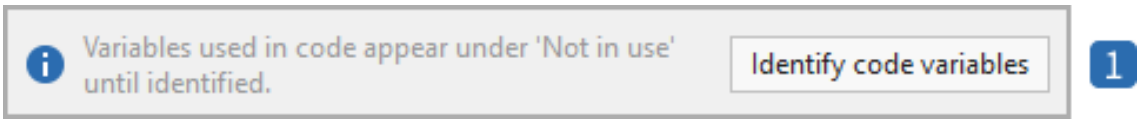
Data binding pad

When opened in the recording module view, the data binding pad shows the status of action, repository, or code variables in this recording module and their default values. The

data binding pad reflects the icons and status designations from the VARIABLES… dialog. These are explained in the next topic below.

1. While in the recording module view, **click** the **View data-binding** button in the toolbar.
2. To display action or repository variables, their status (In use/In use from repository/In use in code/Not in use) and their default values, **click** anywhere in…
   a. the action table or
   b. the repository.





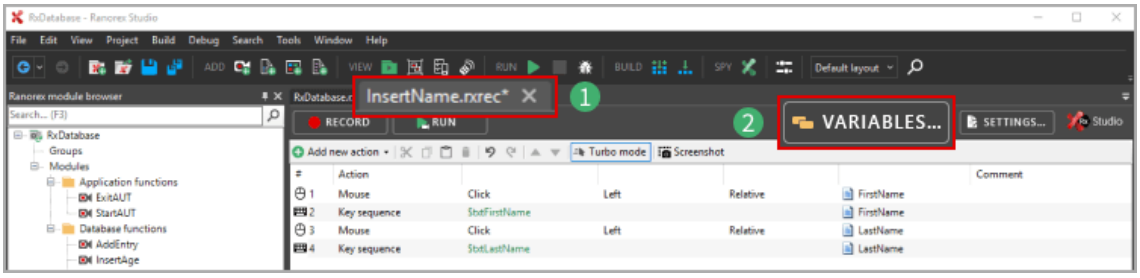
1. If the recording module contains variables that are only used in code, then these initially appear under **Not in use**. The button **Identify code variables** identifies them and causes them to be displayed under **In use in code**.

VARIABLES… dialog

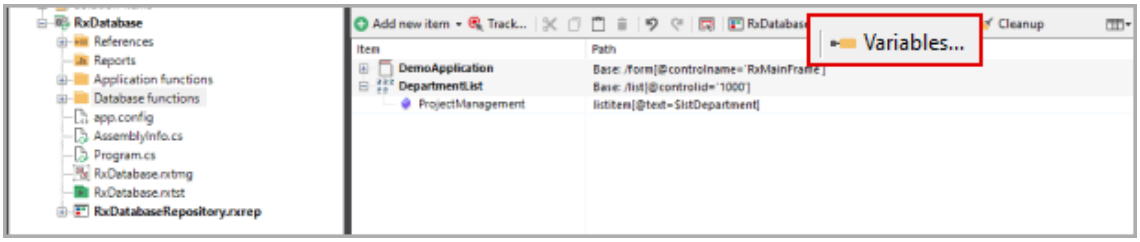
This dialog shows you variable status and also lets you add and edit variables.

To open the dialog for managing variables:

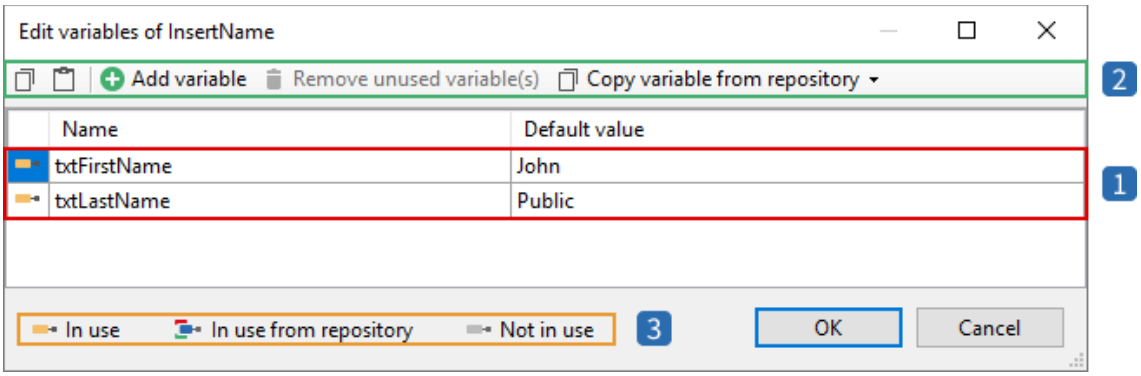
1. For action variables, **open** the recording where they are defined and **click VARIABLES…**.



2 For repository variables, **click Variables…** in the repository view.



The variable management dialog will open:



1 **Variable status, Name,** and corresponding **Default value**

2 ***Variable editor toolbar***

- Options to **copy/paste** a variable to/from the clipboard.
- Option to **add** a new variable directly.
- Option to **remove unused** variables (variable cleanup).
- (Action variable dialog only) Option to **copy a repository variable** to the current recording module, where it becomes an action variable.

3 ***Variable status legend***

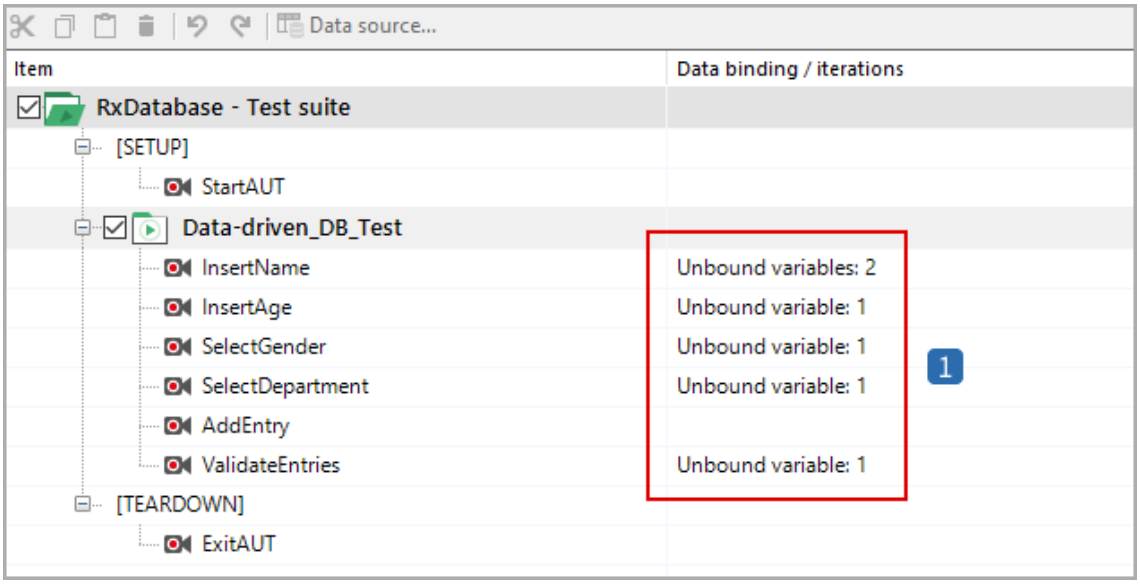
- Variable **in use:** Variable is used, i.e., it is assigned to an action. Does not indicate whether the variable is bound/unbound to a data source.

- (Action variable dialog only) Variable **in use from repository:** Same as previous, but for repository variables that are in use in the action table through a linked repository item.
- Variable **not in use:** Variable is defined, but not assigned to an action item. Does not indicate whether the variable is bound/unbound to a data source.

Overview of defined variables

We've now defined all the variables we need for our data-driven test. When you switch to the test suite view, you can now see the number of variables defined per module. Note they are still unbound, i.e., they have not been assigned a data source.

In the next chapter, we'll define this data source and assign it to our test case. We'll also discuss the options for managing data sources.



1 The test suite view displays the defined, but currently unbound variables

Manage and assign data sources

Data sources are a key component of data-driven tests. They are the place from which variables get their values. In this chapter, we'll find out how to add, manage and assign data sources to test containers so that their data is available for the test run.

Create your data source

Creating a data source is as easy as creating a table. How exactly your data source will look depends on your tests, so making recommendations on how to design your data source is beyond the scope of this user guide.

To follow along with our sample solution, you'll need a data source.

Download the CSV table and unzip it to any directory on your computer or simply copy and paste the following text into a text file and save it as .csv.

```
FirstName,LastName,Age,Gender,Department,Num
John,Public,48,Male,Project Management,1
Mary,Smith,36,Female,Sales,2
Henry,Rogers,29,Male,Support,3
Thomas,Bach,42,Male,Development,4
Cindy,Martens,19,Female,Office,5
Hanna,Perry,48,Female,Management,6
Will,Hallmark,32,Male,Support,7
Nicole,Wallace,38,Female,Testing,8
```

Manage data sources

Data sources, and the test data they contain, are managed per test suite. This means:

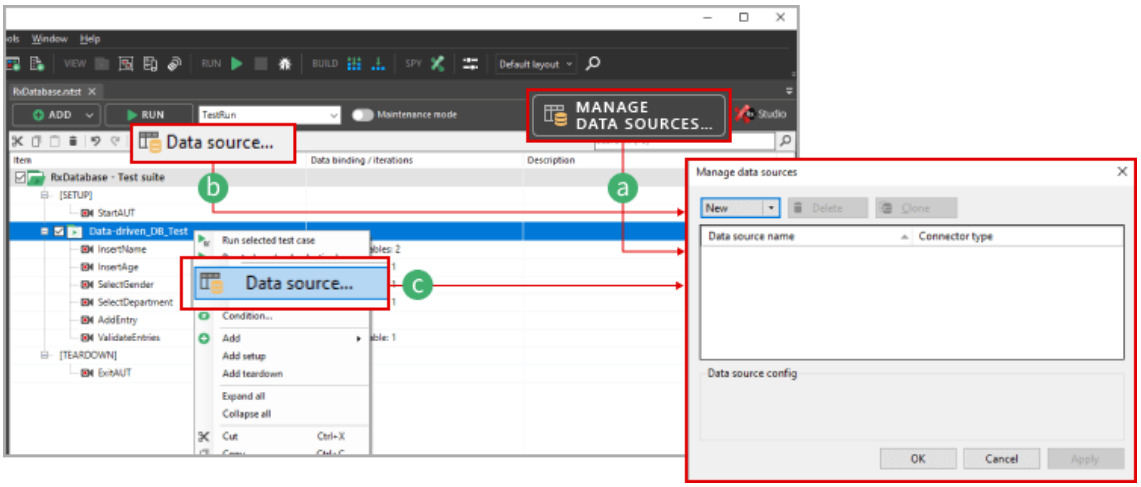
- Once added to a test suite, the data source can be assigned to the test containers of this test suite.
- You can't access a data source in test suite A from test suite B. You first need to add the data source to test suite B.
- A data source can be used in multiple test suites at the same time if you add it to each test suite.

To access data source management directly:

a. **Click MANAGE DATA SOURCES…** in the test suite view.

You can also access data source management through the **Data source…** dialog. With a test container selected:

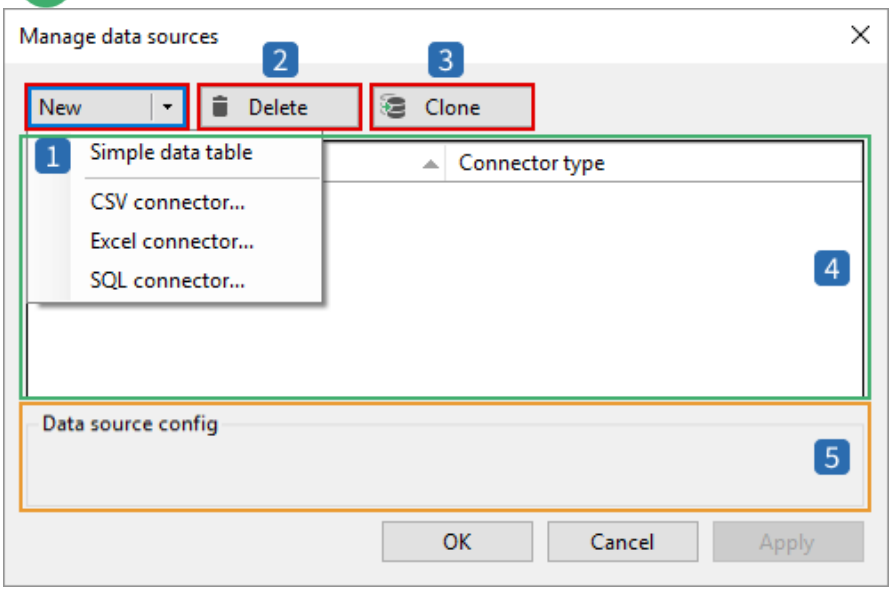
b. **Click Data source…** in the test suite toolbar and then **Manage data sources…**
c. **Click Data source…** in the context menu of a test container and then **Manage data sources…**



The data source management dialog

The data source management dialog appears as shown below. To add the CSV data source for our sample project:

1. **Click New > CSV connector…**



1. **Add** a new data source. Four different types are available, as explained below.
2. **Delete** an existing data source.

Attention

Deleting a simple data table **means deleting the data physically**. **Data will be lost!** This is because simple data tables are stored directly in the corresponding test suite file.

Deleting an Excel, CSV, or SQL data source in this dialog **means deleting only the connector** to this data source and the settings in the Configuration section, not the data source file itself.

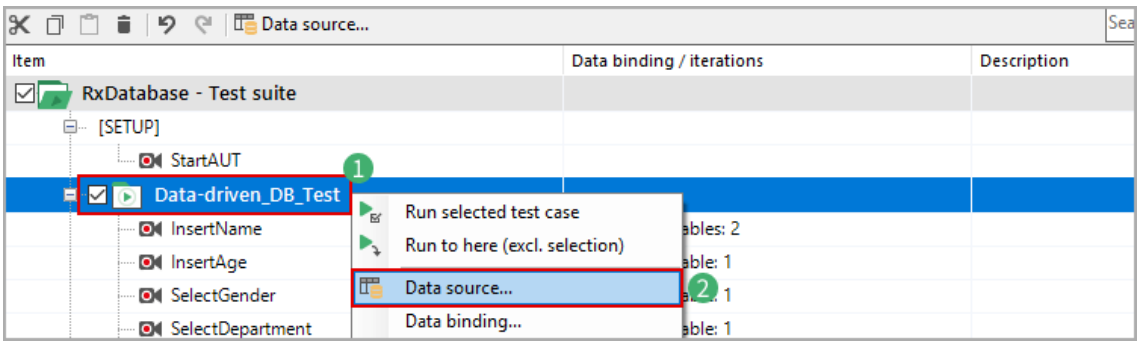
1. **Clone** a data source.
   For simple data sources, this means cloning the contents of the data source and the settings in the Configuration section. For all other data sources, this means cloning the connector and the settings in the Configuration section. This option is useful for specifying different sheets of the same Excel data source, for example.
2. **List of added** data sources, each showing the connector type (Simple, CSV, Excel, SQL) and the Use count (how many times the data source has been assigned in the test suite).
3. **Configuration** section. Here you can manage certain settings depending on the data source type. The available settings are explained below for each data source type.

Assign data sources

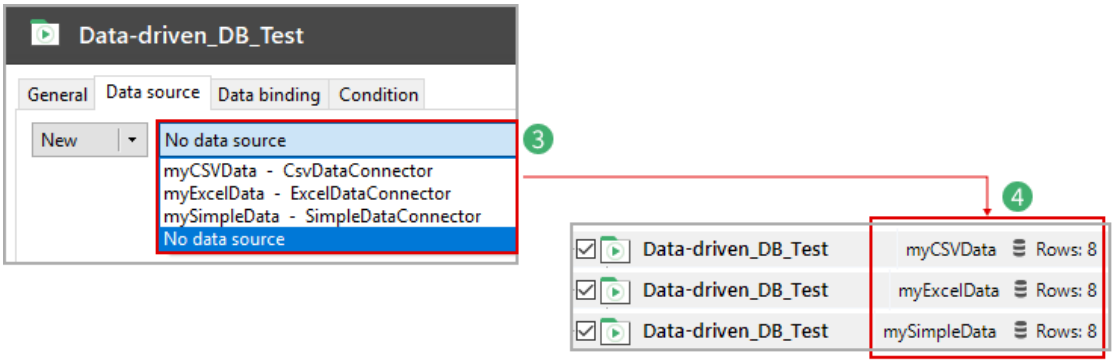
Once you've added a data source, you then need to assign it to a test container, so the modules/variables in it can access the data.

To do so:

1. **Select** the test container you want to assign a data source to.
2. **Open** the context menu and **click Data source…**



3. **Select** the desired data source from the drop-down menu and click OK.
4. The data source appears next to the test container in the test suite view, with the number of rows in it indicated.



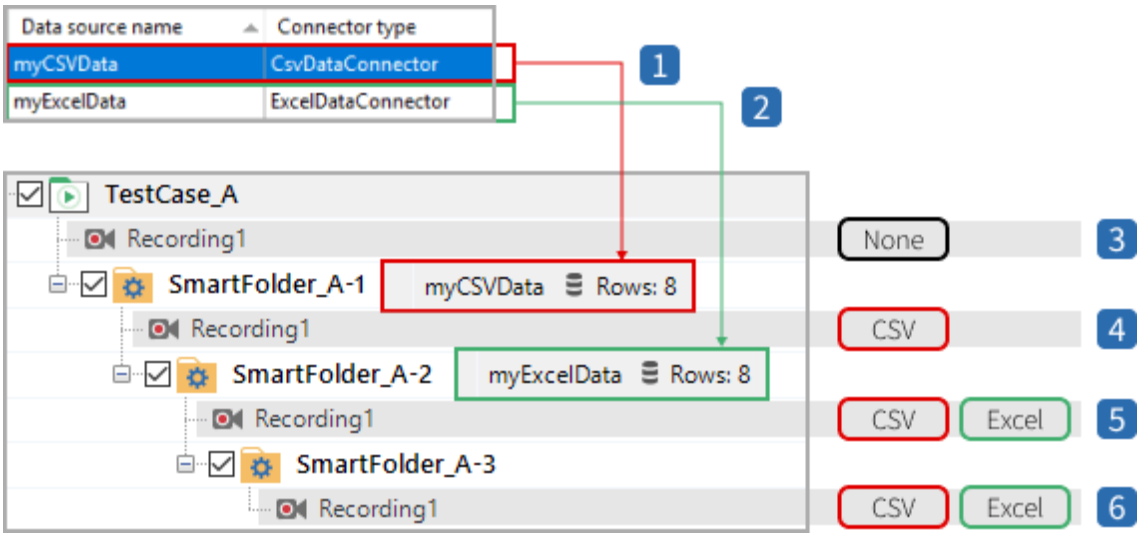
Data source assignment rules

The assignment of data sources to test containers is subject to the following three rules:

Rule 1	Once assigned to a test container, a data source cannot be assigned to descendants of this test container.
Rule 2	Once assigned to a test container, the contents of the data source can be accessed by all descendants of this test container, but not by its siblings or ancestors.
Rule 3	Multiple assigned data sources in a tree complement each other; they do not replace each other.

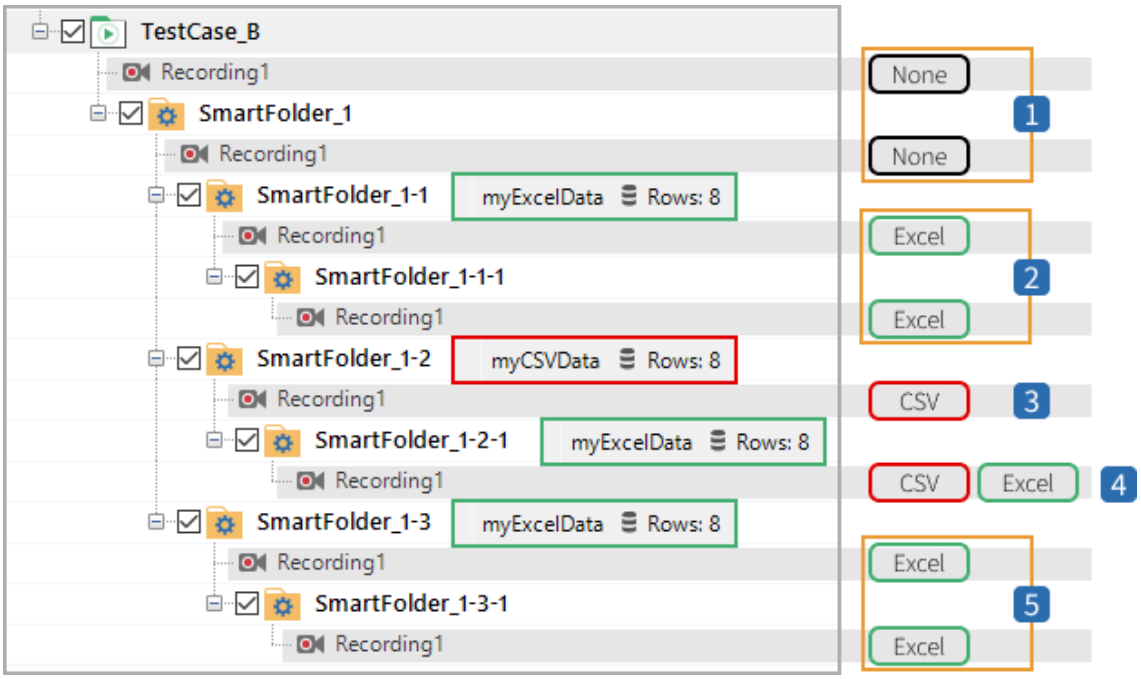
Example 1

Let's see how these rules work by way of an example. Suppose we have two data sources available in a test suite: a CSV data source, **myCSVData**, and an Excel data source, **myExcelData**.



1. The CSV data source is assigned to the smart folder **A-1.**
2. The Excel data source is assigned to the smart folder **A-2.**
3. Because of rule 2, test case A and its module cannot access the data sources of the descendant test containers.
4. Because of rule 2, modules of the smart folder **A-1** have access to the CSV data source.
5. Because of rules 2 and 3, all modules of the smart folder **A-2** have access to both the CSV and Excel data sources.
6. Because of rules 2 and 3, all modules in descendants of the smart folder **A-2** have access to both the CSV and Excel data sources.

Example 2



1. All data sources have been assigned below test case **B** and smart folder **1**. Therefore, these test containers and their modules cannot access the data sources.
2. The Excel data source is assigned to the smart folder **1-1**. Therefore, the descendants (modules, test containers) of this tree branch can access the Excel data source.
3. The CSV data source is assigned to the smart folder **1-2**. Therefore, its modules can access this data source.
4. The Excel data source is assigned to smart folder **1-2-1**. Therefore, its descendants can access both the CSV and the Excel data sources.
5. The Excel data source is also assigned to smart folder **1-3-1**. Therefore, its descendants can access this data source.

Data source types and connectors

Ranorex Studio supports four different types of data sources: Simple, CSV, Excel, and SQL data tables.

With the exception of simple data tables, all of these sources are added via **connectors.** This means that Ranorex Studio only links to the data table file. It does not add the contents of the file to the test suite.

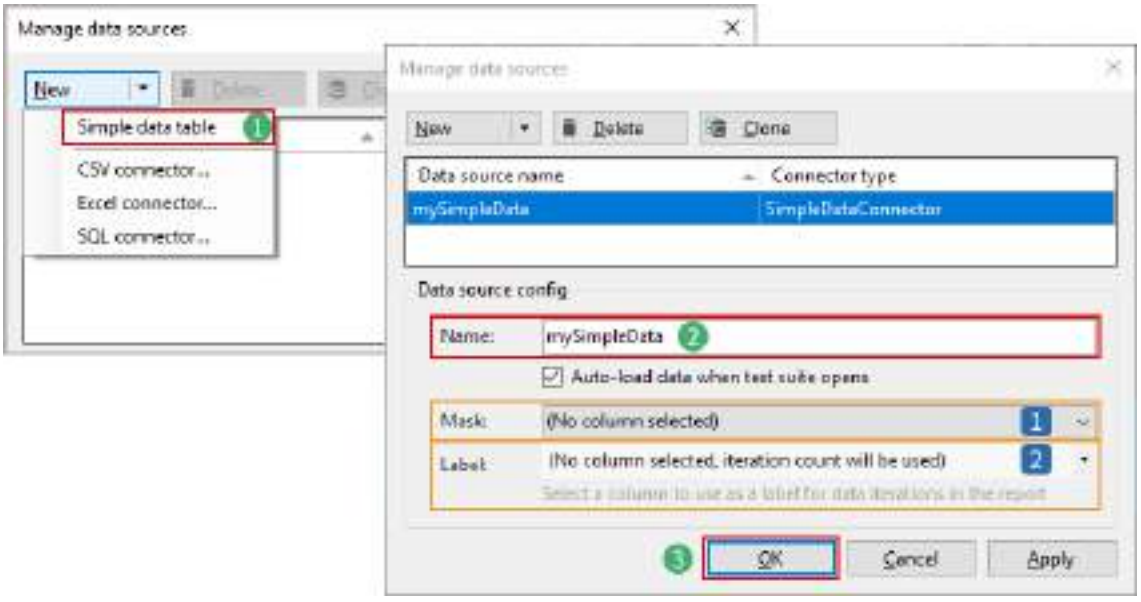
Simple data table

Simple data tables are useful for when you want to quickly set up small data-driven tests, e.g. for trial and error. We do not recommend them for anything that's more complex than a couple of data rows.

Simple data tables are stored directly in the test suite file (.rxtst), with all of their contents. This is why you have to create and maintain them directly in the **Data source…** dialog. It's also why they are deleted entirely when you delete them in the data source management dialog, unlike other data sources.

To add a new simple data table:

1. **Click New** > **Simple data table.**
2. **Name** the data source.
3. **Click OK.**

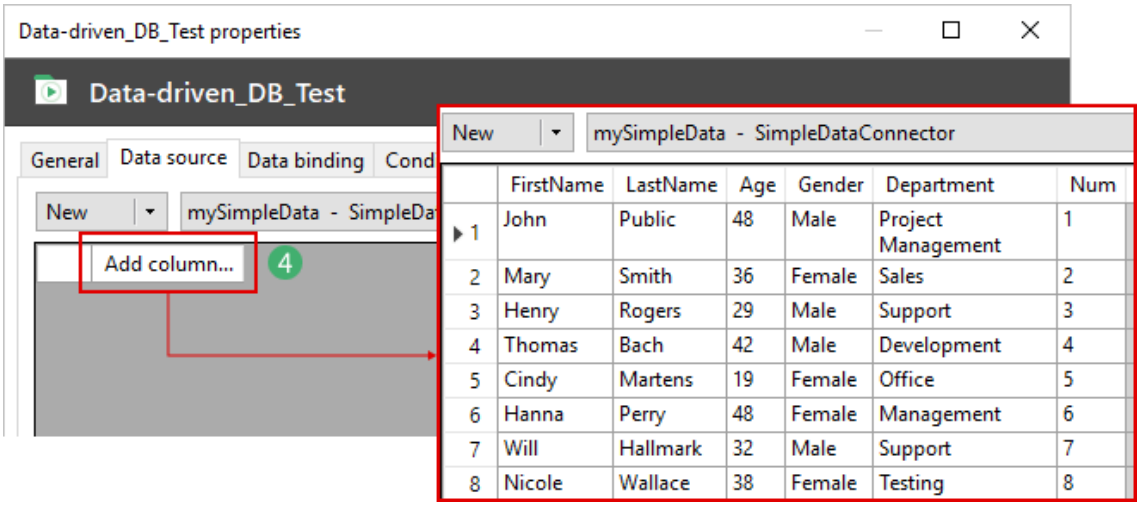


1. The **Mask** option is explained separately further below.
2. The **Label** option is explained separately further below.

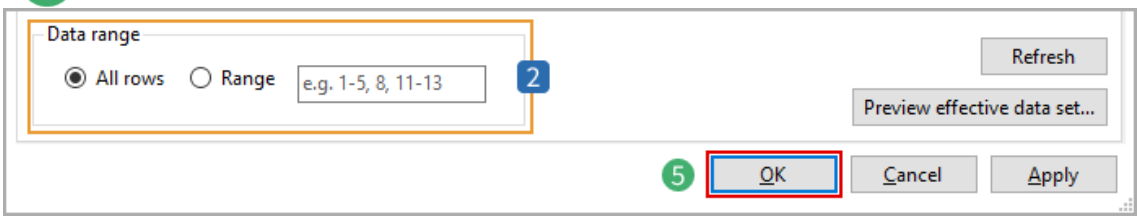
4. In the **Data source…** dialog of a test container, **select** the simple data source and create the content in the table editor.

Hint

You can paste tables from Excel files into the table editor.



5 **Click OK** when you're done.



2 You can also specify a data range. This option is explained separately further below.

Excel data connector

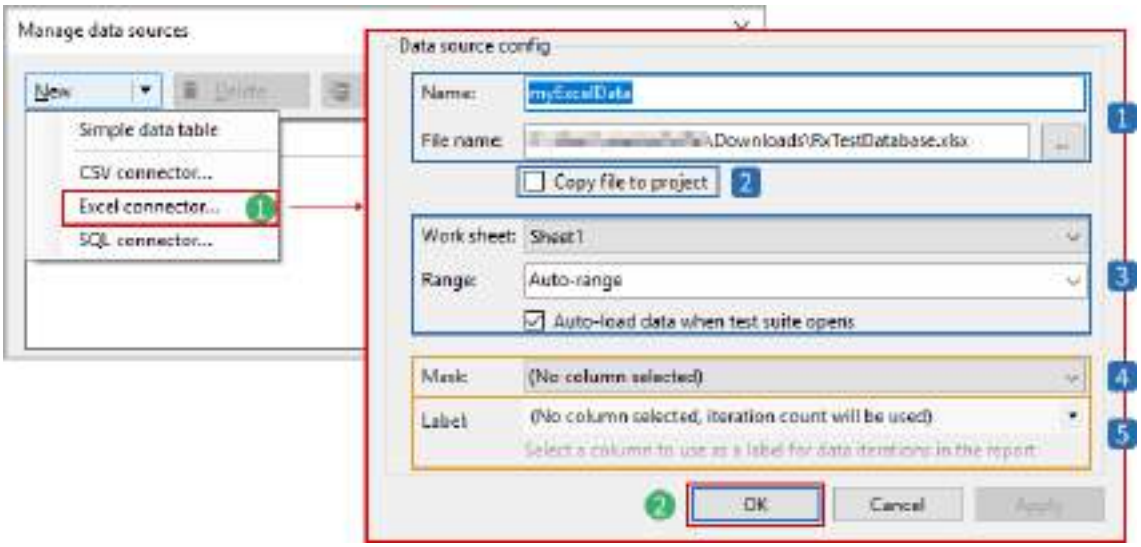
Excel data sources are added via a connector.

Hint

Instead of the default Excel file format `xlsx`, you can also use the **native binary file format** `xlsb`. This file format is supported in Microsoft Office 2007 and later, and is much faster than the non-binary version.

To add an Excel connector:

1. **Click New > Excel connector…**
2. **Configure** the connector and click **OK.**



Excel connector configuration:

1. Name **the Excel connector and** specify **the location of the Excel file.**
2. **This will copy the Excel file to your project folder. You must check this if you use** version control**.**

3. **Work sheet selection**

If your Excel file contains more than one worksheet, you can specify the sheet to be used here. You can also limit the test data to a specific range.

Uncheck the auto-load option to decrease the start-up loading time for the test suite. However, this also means the number of rows won't be displayed next to test containers.

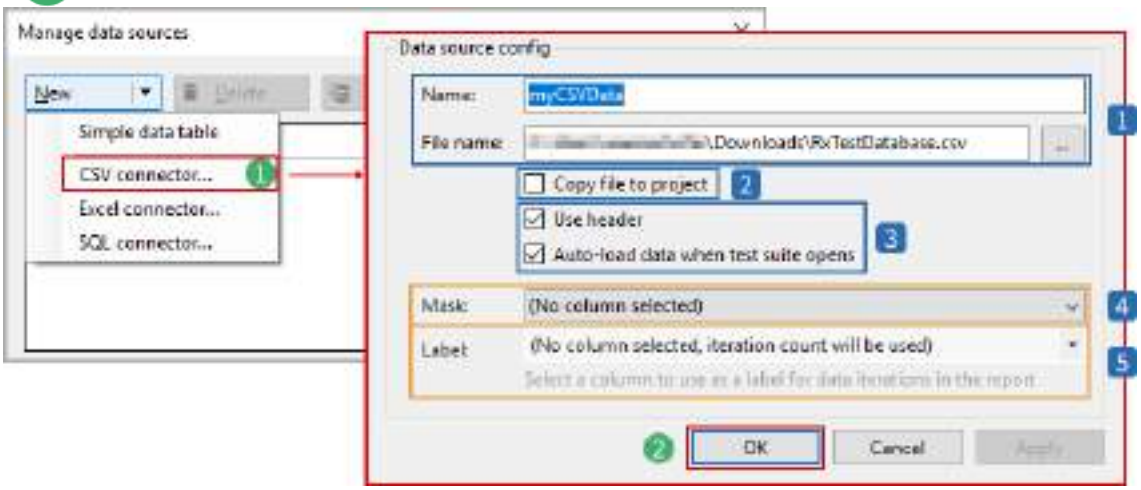
4. The **Mask** option is explained separately further below.
5. The **Label** option is explained separately further below.

CSV data connector

CSV data sources are added via a connector. Once added, CSV data sources can be edited in Ranorex Studio in the **Data source...** dialog. When you save these changes by clicking **OK** or **Apply**, the actual CSV file will also be changed.

To add a CSV connector:

1. **Click New > CSV connector...**
2. **Configure** the connector and **click OK**.



CSV connector configuration:

1. **Name** the CSV connector and **specify** the location of the CSV file.
2. This will copy the Excel file to your project folder. You must check this if you use **version control**.
3. **Data configuration**

Specify whether the CSV file contains a header row or not.

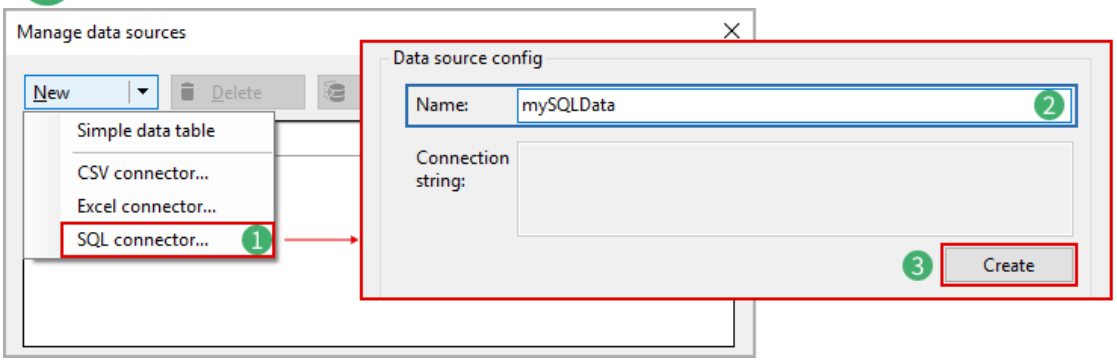
Uncheck the auto-load option to decrease the start-up loading time for the test suite. However, this also means the number of rows won't be displayed next to test containers.

4. The **Mask** option is explained separately further below.
5. The **Label** option is explained separately further below.

SQL data connector

With the SQL data connector, you can access an SQL database and pull data from it using an SQL query. We'll illustrate this process with a simple example where we access a Microsoft Access database.

1. **Click New > SQL connector…**
2. **Name** the connector.
3. **Click Create** to specify the SQL connection string.



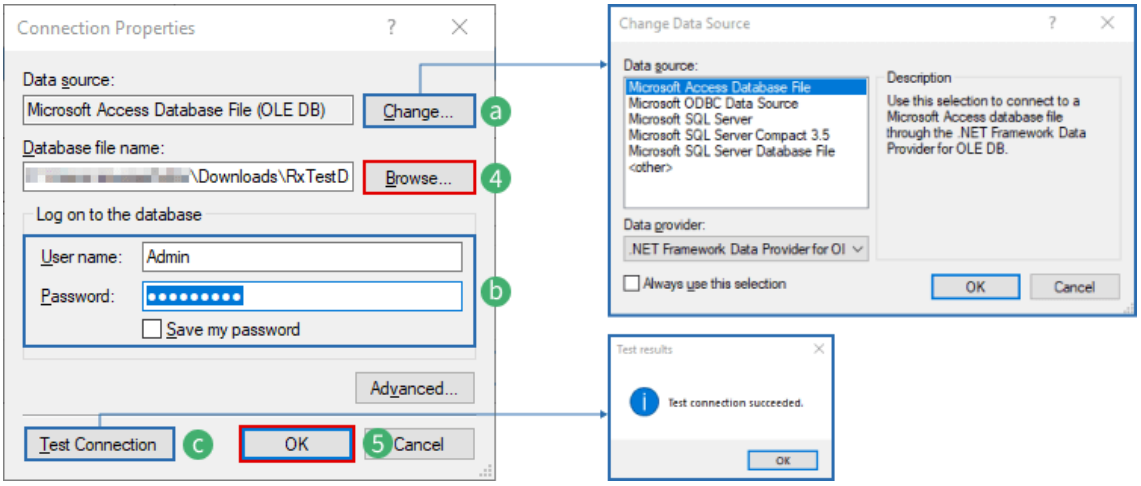
4. **Specify** the location of the database file.
5. **Specify** the optional connection settings (see below) and **click OK**.

Optional connection settings:

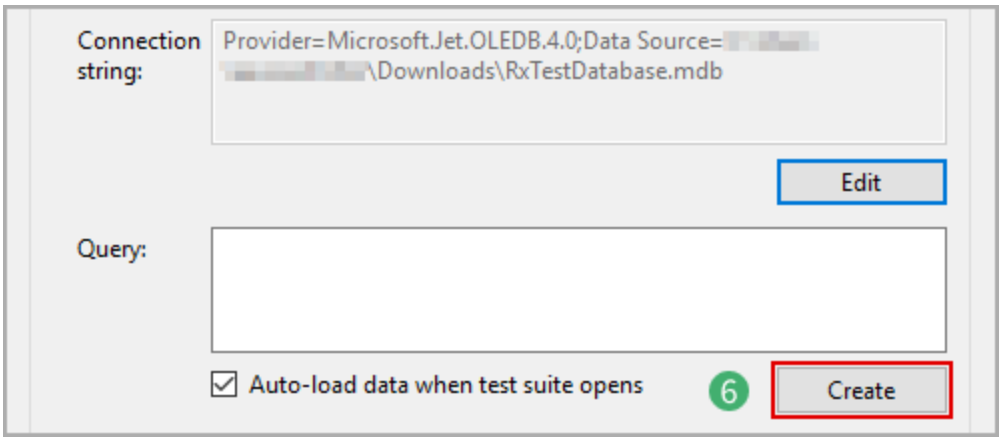
a. **Change** the database connection type to suit your database type. In our example, Microsoft Access Database File is correct, since we're using a Microsoft Access database.

b. If the database requires a login, **specify** it here.

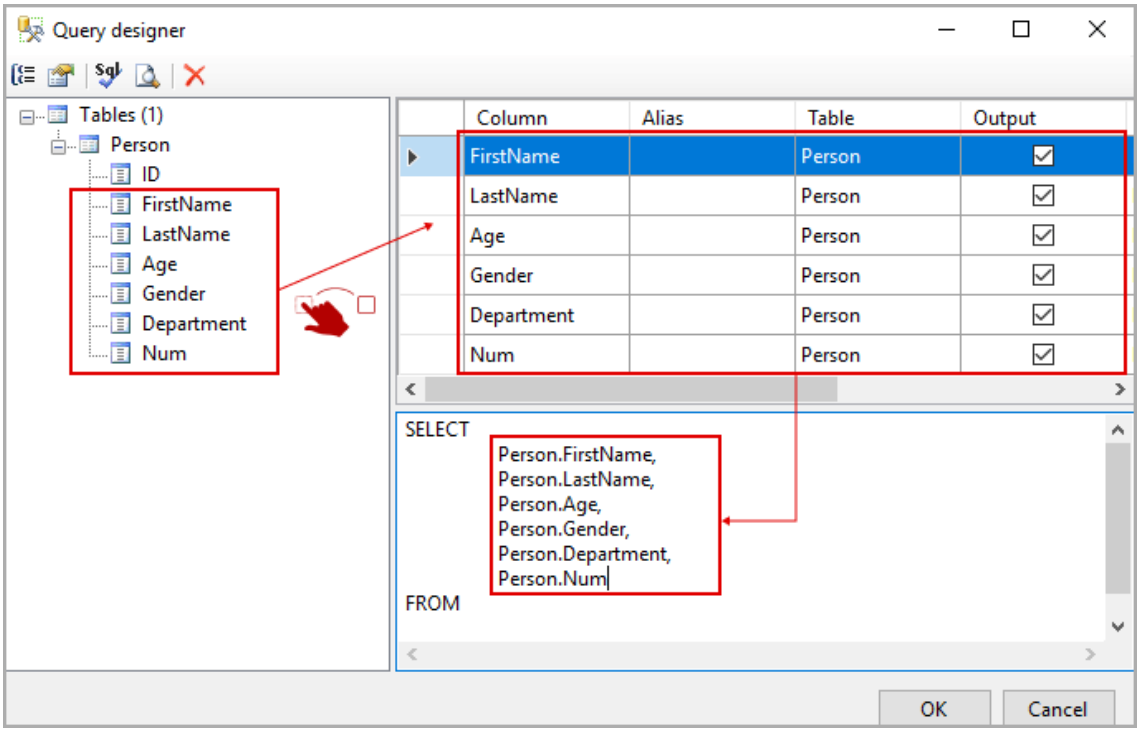
c. **Click** to test the connection to the database (recommended).



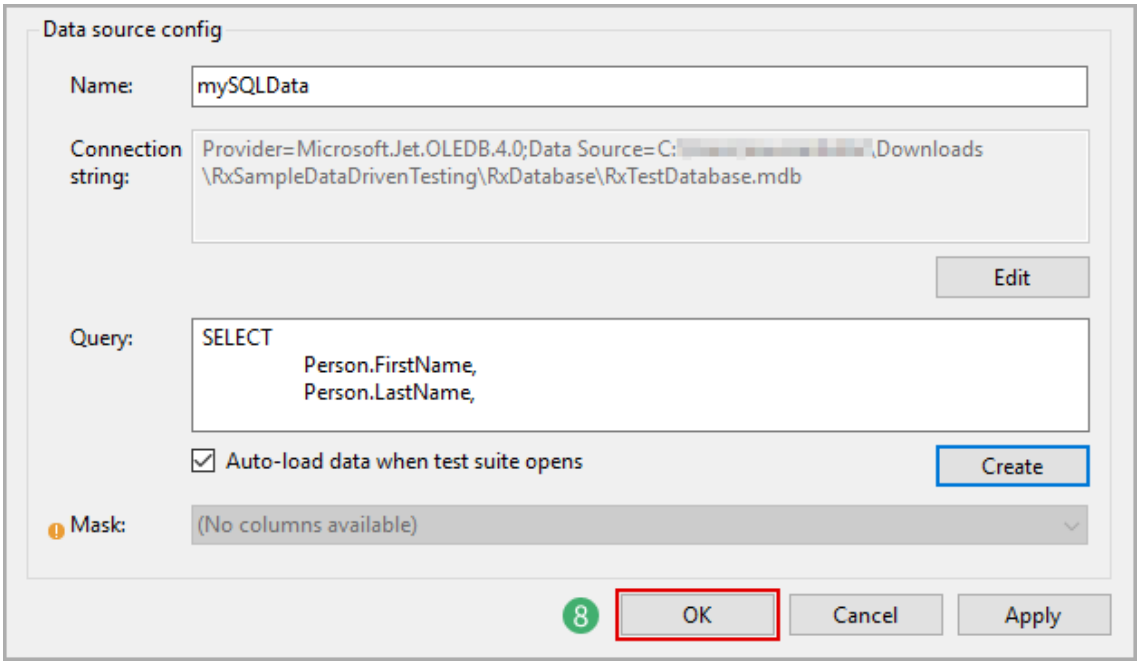
6. Under Query, **click Create** to specify the database query.



7 **Define** the desired SQL query in your database (Microsoft Access, in our example) to provide the data to Ranorex Studio and **click OK.**

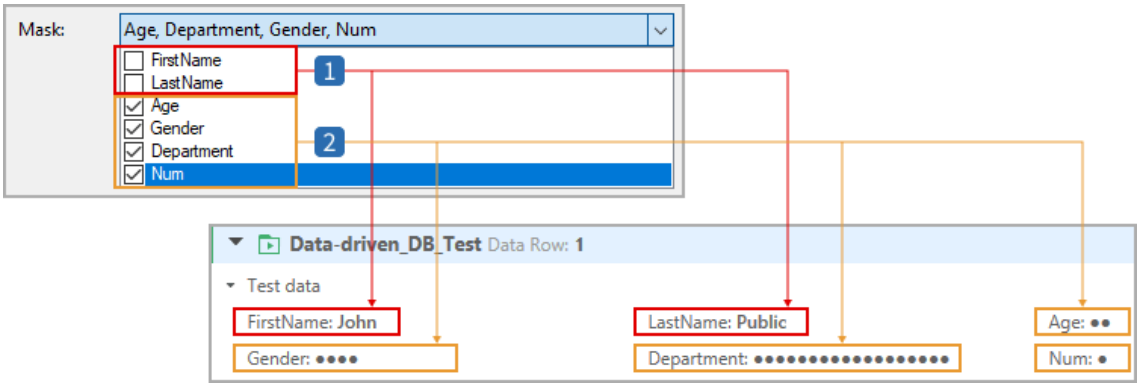


8 **Set** the auto-load behavior (disable for faster load time, but missing row indicators in the test suite view) and masking (explained separately below) and **click OK**.



Mask data

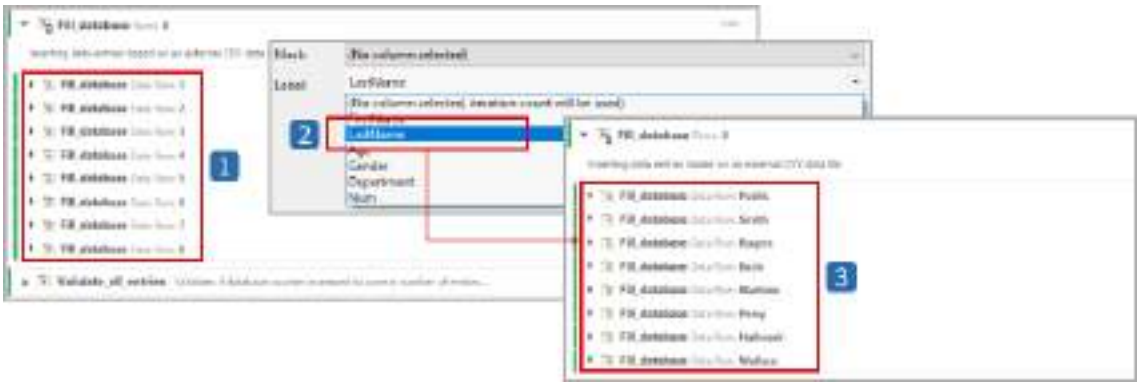
You can mask data for all data source types. This way, you can hide sensitive data in the report while still allowing Ranorex Studio access to it for testing purposes.



1. Data from **unmasked columns** is displayed normally in the report.
2. Data from **masked columns** is hidden in the report.

Label data iterations

By default, data iterations carried out as part of a data-driven tests use the iteration count as a label in the report, i.e. they are numbered. However, you can also assign column names from your data source as labels. Ranorex Studio will then use the values in that column as labels in the report.

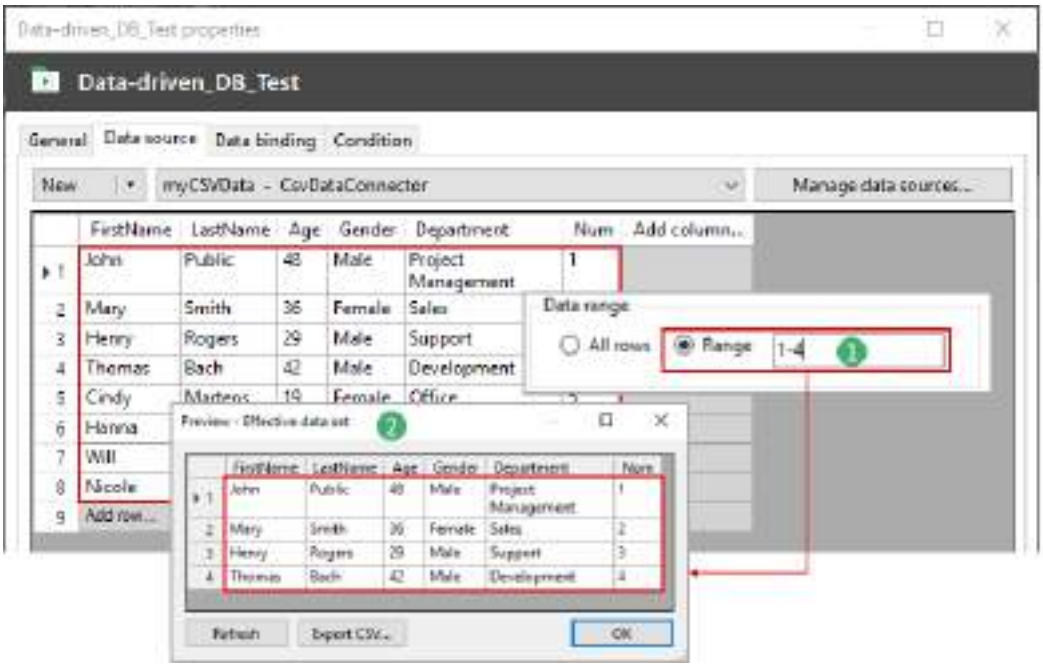


1. Default iteration count with numbers from 1 to 18 in the report.
2. Data-source column **LastName** selected as the label instead.
3. The iterations now use the respective values of the column **LastName** as label, making them easier to identify.

Limit data range

You can limit the data range for all data sources. This allows you to make only certain rows of a data source available to a test container.

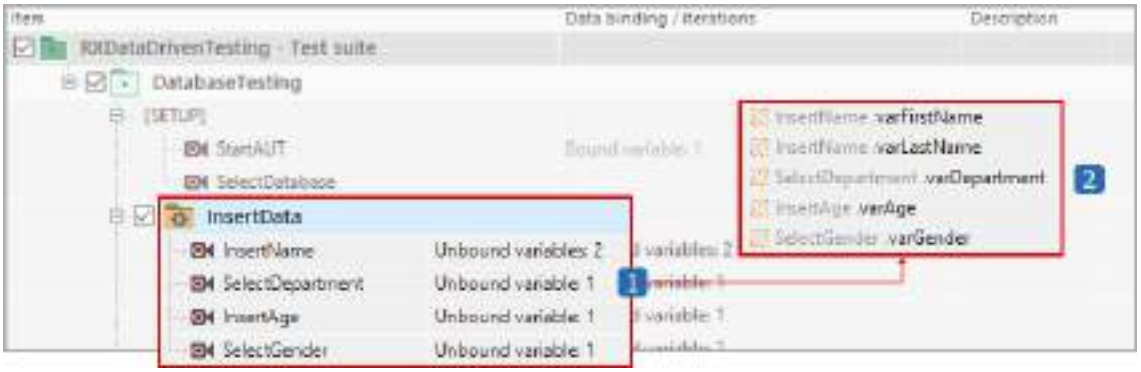
1. **Enter** a range of rows in the **Data source…** dialog.
2. **Click Preview effective data set…** to see the result.



Auto-create a CSV data source from variables

You can automatically create a CSV data source from unbound variables defined in modules.

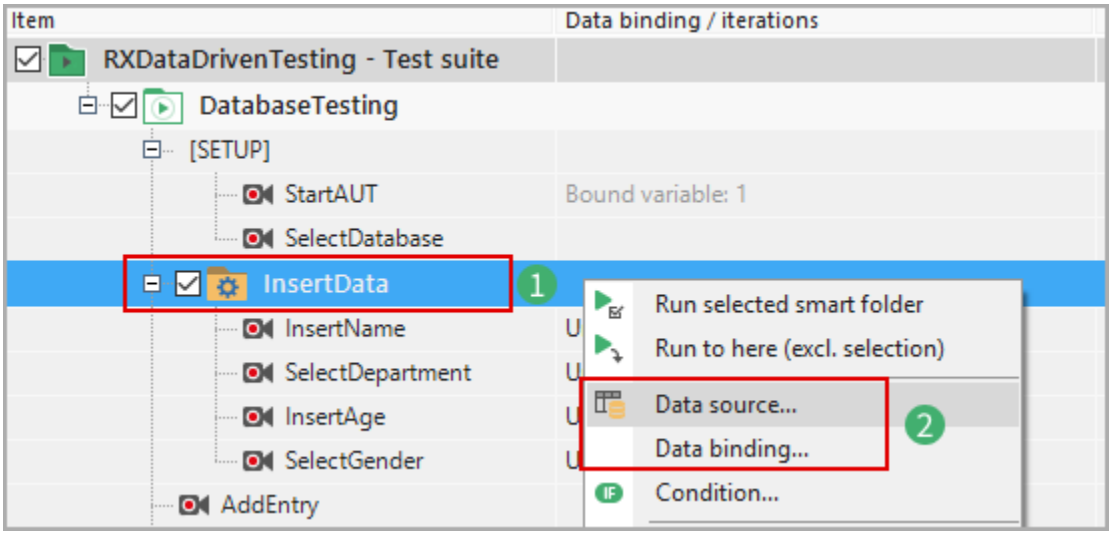
The test suite shown below contains the smart folder InsertData, which in turn contains four recording modules with 5 defined, but unbound variables.



1. Smart folder **InsertData** with four recording modules.
2. The 5 defined, but unbound module variables in the recording modules.

To auto-create a CSV data source from these variables:

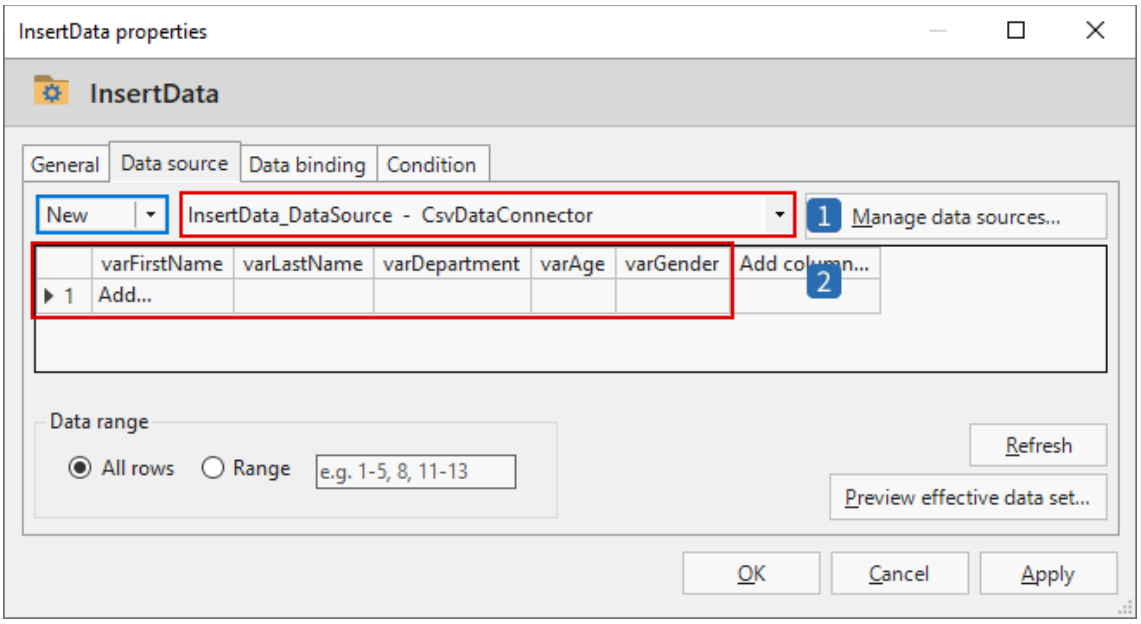
1. **Right-click** the smart folder.
2. **Click** either **Data source…** or **Data binding…**.



3 **Click Auto-generate data source**.

Result

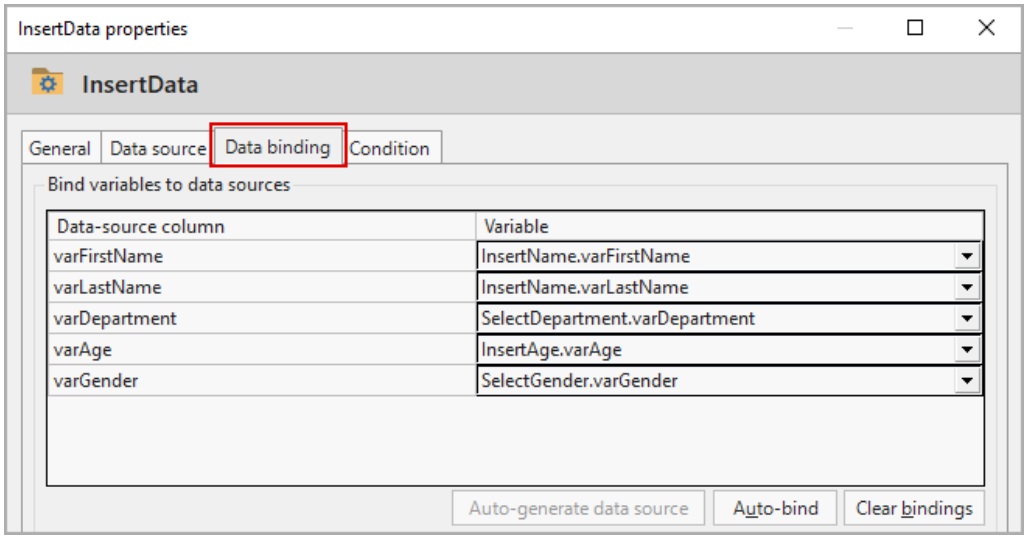
Ranorex Studio automatically creates a CSV data source with the variable names as the column names.



1 Auto-generated CSV data source with the default name derived from the test container.

2 Table with the column names derived from the varible names.

Ranorex Studio also automatically binds the variables to the matching columns in the auto-created data source, visible under Data binding.



You can manage the data source as usual in the Manage data sources… dialog. By default, the CSV file is stored in the project folder of your solution.

1. Auto-generated CSV data source in the Manage data sources… dialog.
2. The CSV file in the project folder of your solution.

Summary of the steps for this chapter

Now that we've explained all the options for managing and assigning data sources, let's quickly go through the required steps to prepare our sample solution for the next chapter again.

1. **Ensure** you have your data source ready (CSV file).
2. In the test suite view, **click MANAGE DATA SOURCES...**.
3. **Click New > CSV connector...**
4. **Name** it **myCSVData**, **specify** the location of the file, **check** all three boxes, and **click OK**.
5. In the test suite view, **right-click** the test case **Data-driven_DB_Test** and **click Data source...**
6. From the drop-down menu, **select myCSVData** and **click OK**.

You've now assigned the data source. Your solution is ready for the next step: data binding.

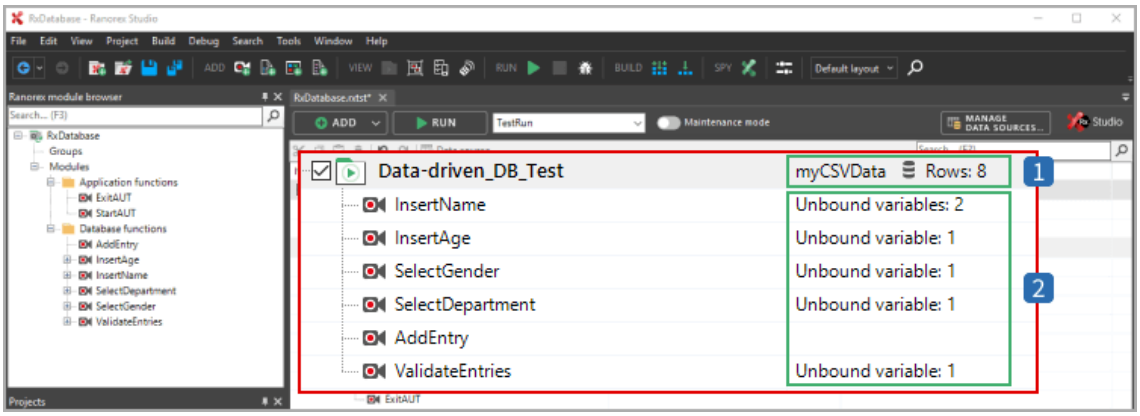
Data binding

Now that we've defined our variables and assigned the data source to the test case, we're ready to connect them. This is called **data binding,** which is the topic of this chapter.

> **Screencast**
> The screencast "Data binding" walks you through information found in this chapter.:
>
> Watch the screencast now

Initial situation

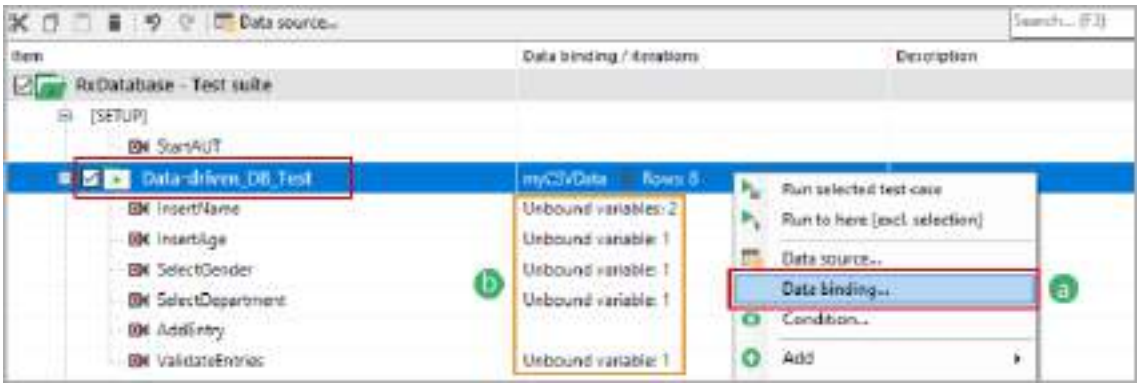
The below image shows the initial situation with the data source assigned to the test case and the defined, but unbound variables next to the recording modules.



1. Data source **myCSVData** with 8 data rows assigned to a test case.
2. 6 unbound variables in 5 recording modules.

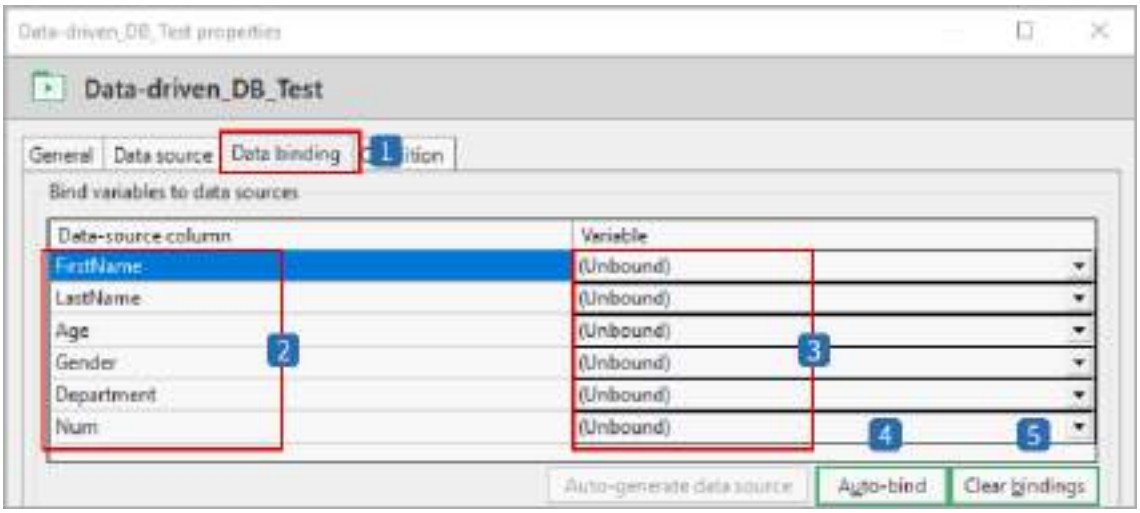
Access data binding

1. **Use** one of the following options**:**
   a. **Right-click** a test container and **click Data binding…**
   b. **Double-click** an unbound variable.



The data binding dialog

The data binding dialog opens with the following items:



1. **Data binding** tab in the properties window of the test container
2. List of available **data columns** from the assigned data source
3. Multiple drop-down menus containing the defined **variables** available for binding
4. **Auto-bind** function, explained at the end of this chapter
5. Unbinds all variables from their data columns

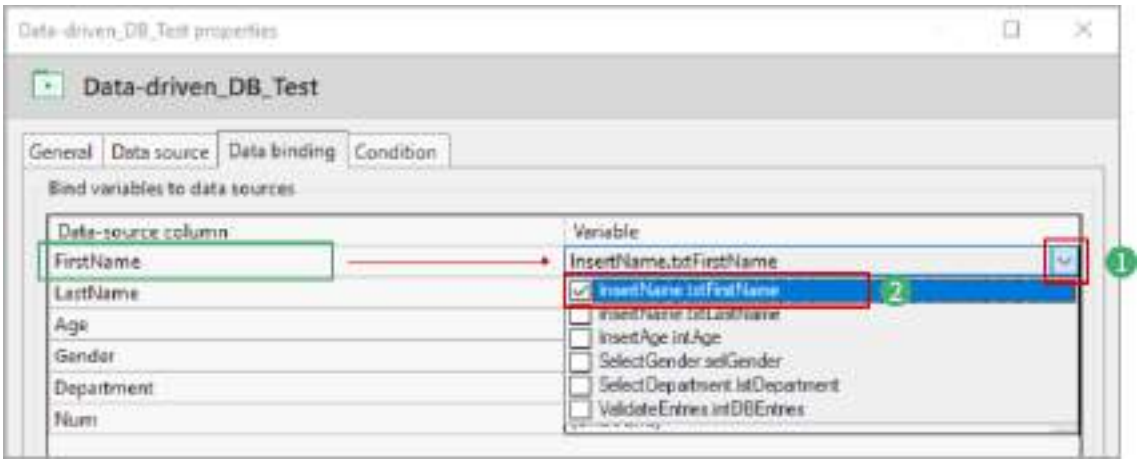
Bind data to variables

Binding data to variables means specifying one variable per column.

For the column **FirstName**:

1. **Open** the drop-down list of variables next to the data column.
2. **Check** the variable to which the data will bound. In this case: **InsertName.txtFirstName**

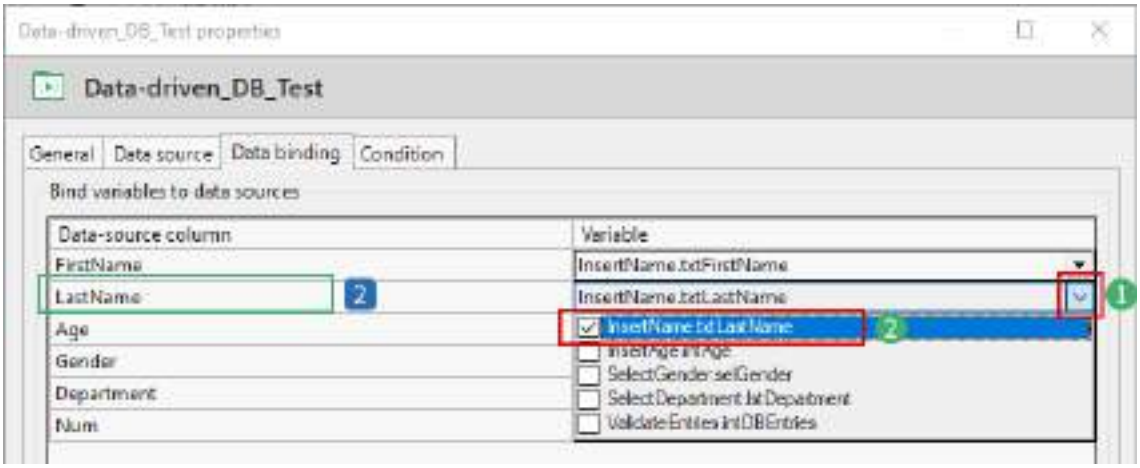
> **Note**
>
> Variable names in the data binding dialog are a combination of the module they are defined in and their actual name without the preceeding $. So, for **InsertName.txtFirstName**, **InsertName** is the recording module, then there is a separating period, and **txtFirstName** is the actual variable name.



For the column **LastName**:

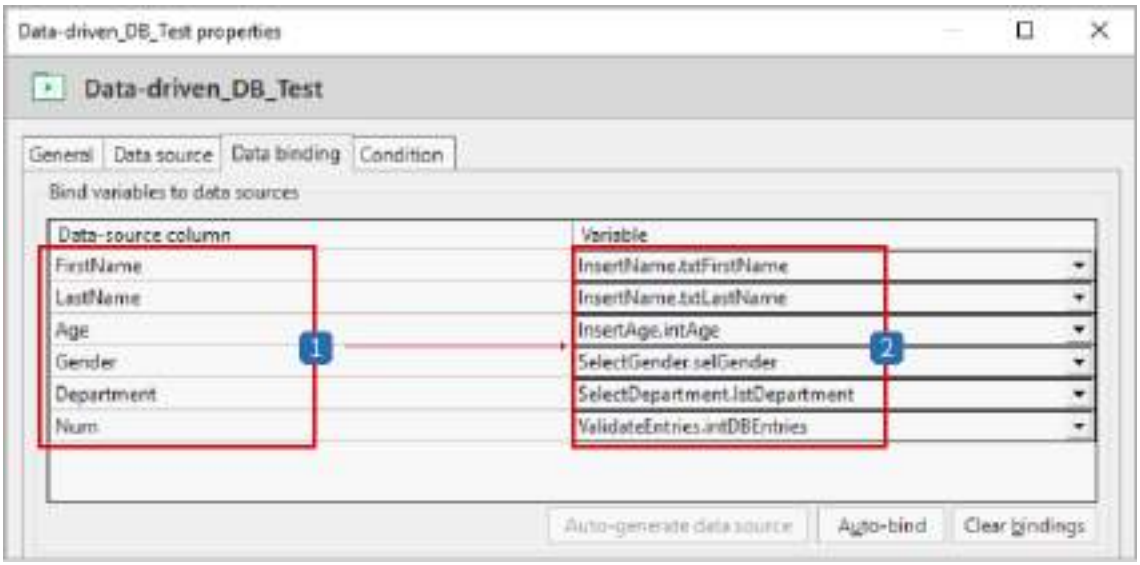
1. **Open** the drop-down list of variables next to the data column.
2. **Check** the variable to use for the data binding. In this case: **InsertName.txtLastName**

> **Note**
>
> Only unbound variables are displayed. **InsertName.txtFirstName** is missing because we've already bound the column **FirstName** to it.



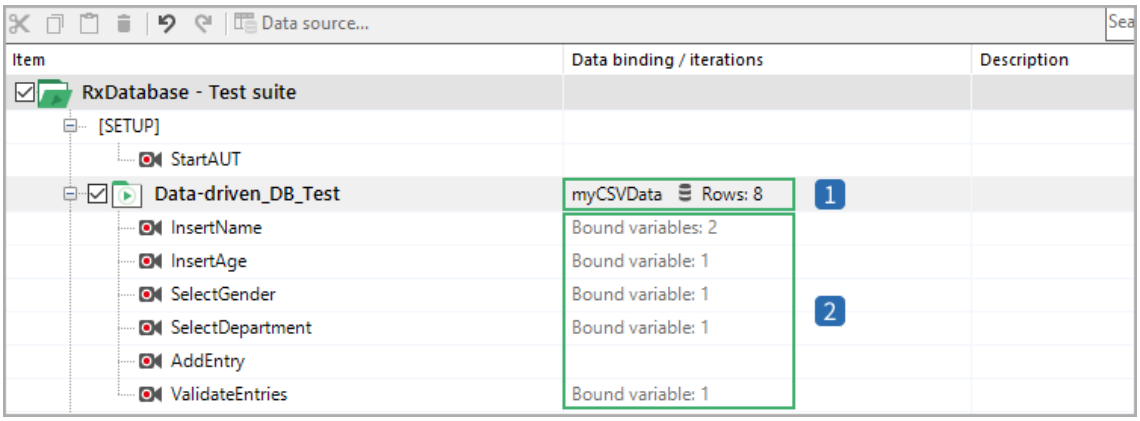
Repeat these steps for the remaining columns/variables.

Result



1. List of available **data columns** from the assigned data source.
2. List of bound **variables.**

Resulting test suite view



1 Data source **myCSVData** with 8 data rows assigned to a test case.

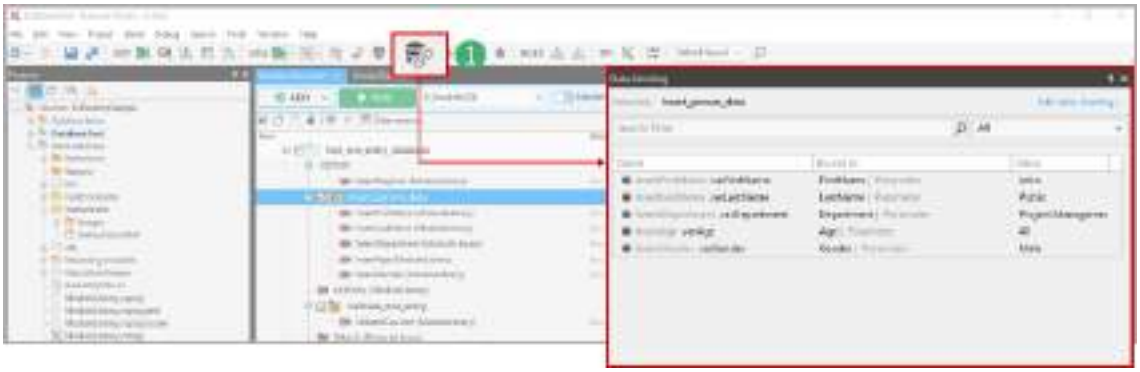
2 6 bound variables in 5 recording modules.

Our data-driven test is now complete. In the next chapter, we'll run it and get our report.

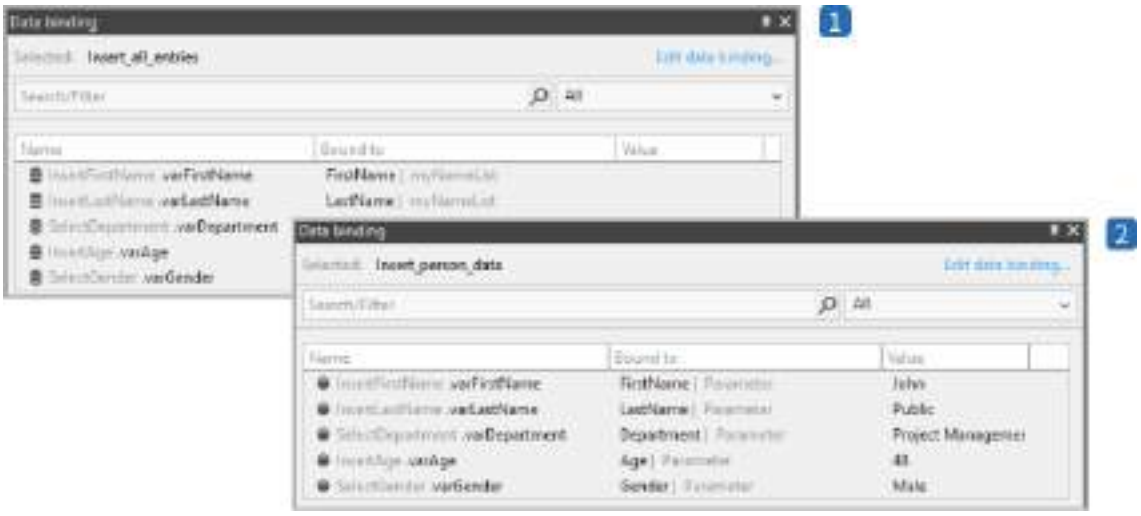
View status in data binding pad

The data binding pad gives you a quick and detailed overview of a test container's data binding, i.e., its variables and parameters, their status, and their values.

1 To open the data binding pad, **click** the **View data binding** button in the toolbar and **select** a node (i.e., an item) in the test suite.



2 The data binding pad displays the data binding information for this node.



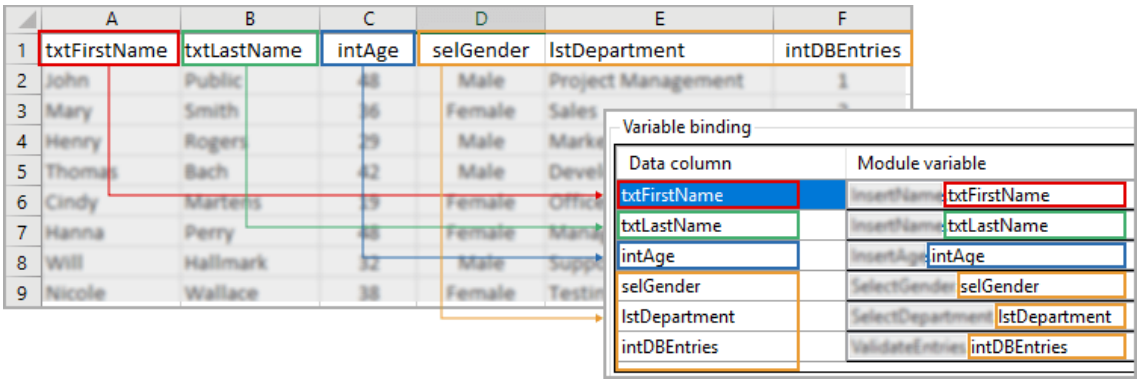
1 Data binding pad showing variables bound to a data source.

2 Data binding pad showing variables bound to parameters.

Auto-bind

This option automatically binds all variables to data source columns or to parameters of the exact same name.

> **Hint**
>
> Design your variables and data sources to be auto-bindable for quick data binding. An easy way to do so is to create your variables, and then Auto-create a simple data source in the Data-binding dialog.



Auto-bind is available in three places:

- in the data-binding dialog (see further above)
- in the data-binding pad
- in a message displayed when adding/moving a module that has auto-bindable variables

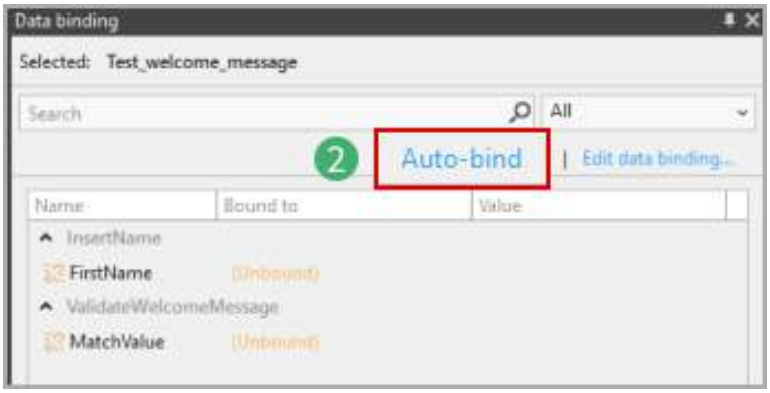
> **Note**
>
> When you auto-bind variables from the data-binding pad or the message and a variable fits both a data-source column and a parameter, **then it is auto-bound to the parameter.**

Auto-bind from data-binding pad

To auto-bind this way:

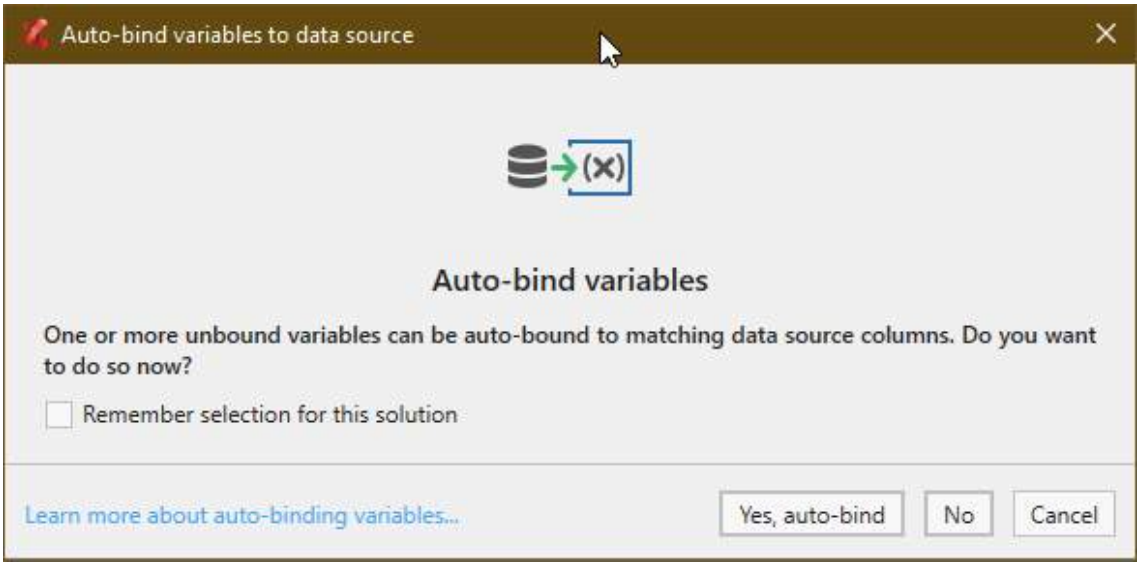
1. **Select** an item with unbound, auto-bindable variables in the test suite.
2. **Click Auto-bind** in the data-binding pad.

> **Hint**
>
> Select the test suite node and click Auto-bind to **instantly bind all auto-bindable variables** in the entire test suite!



Auto-bind from message

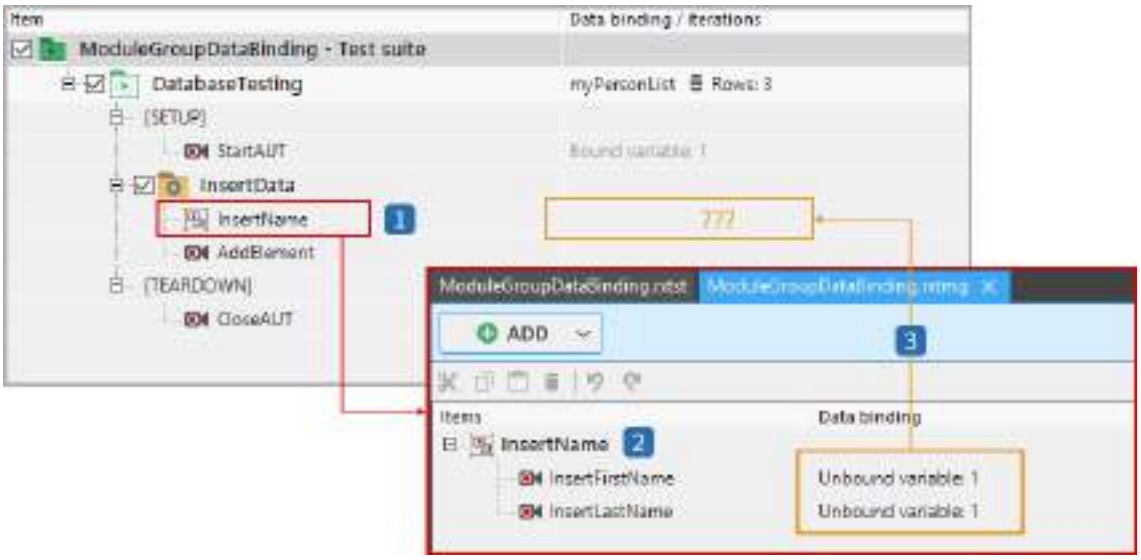
When you add or move a module that contains unbound auto-bindable variables, a message will pop up to ask you if you want to auto-bind these variables. You can also click **Remember selection for this solution** to always do so automatically.



Data binding in module groups

Data binding for module groups works a little differently than for test containers (test cases, smart folders).

Initially, variables used in recording modules that are inside a module group are not visible in the test suite view. You can only see them if you open the module group.



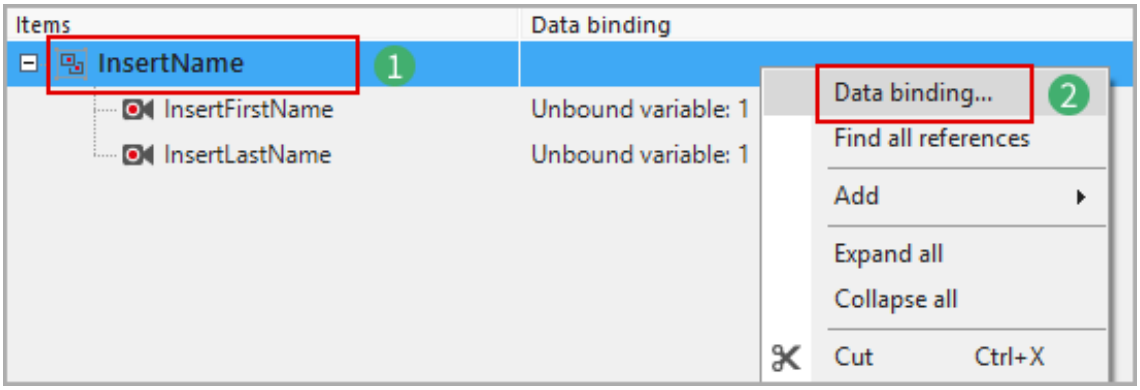
1. Module group **InsertName** in the smart folder **InsertData**.
2. The module group contains two recording modules, each with one defined but unbound variable.
3. These variables are not yet visible in the test suite view and therefore can't be used. At this point, they're only visible inside the module group. They are encapsulated in it.

Bind the variables in the module group to data

To make these encapsulated variables visible in the test suite and bind them to data, we need to first bind them to module-group variables. These act as a bridge from inside the module group to the test container.

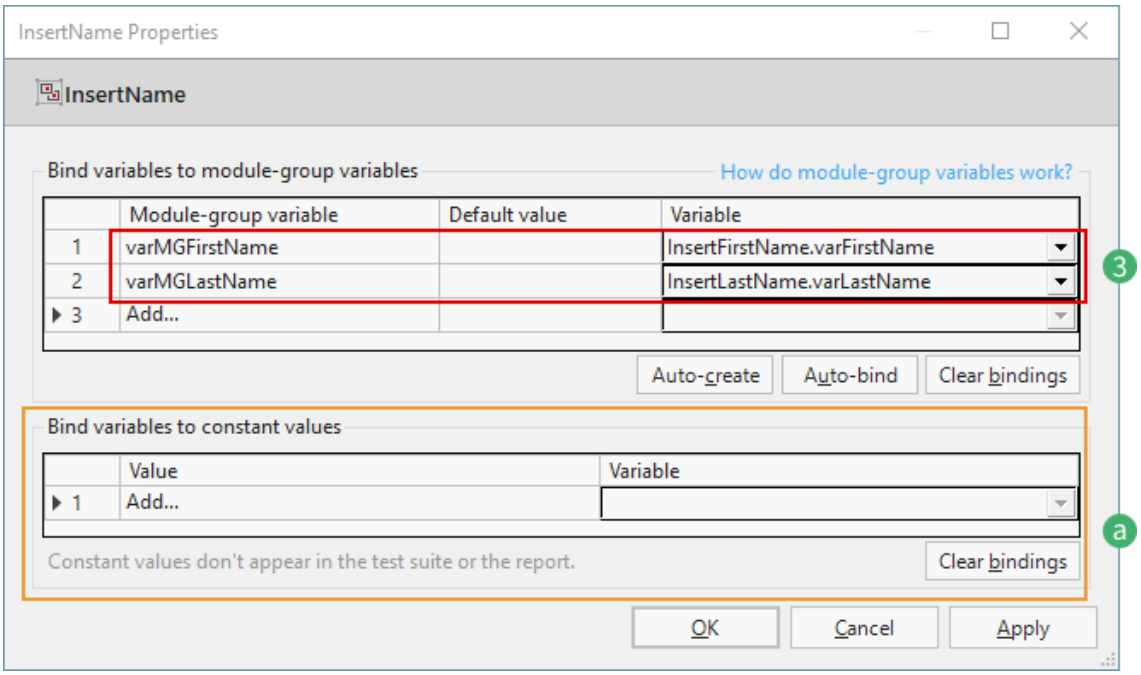
To bind variables to module-group variables:

1. In the module group view, **right-click** the module group.
2. **Click Data Binding...**



3. **Add** a module-group variable for each recording-module variable and **bind** the recording-module variables to the module-group variables.
   a. You can also bind the recording-module variables to constant values. This way, the variables will not be visible/usable in the test suite view, but still be bound to values.

> **Hint**
> Constant values do not appear in the report either. This is useful for binding variables to values that not everyone who reads the reports should be able to see.



Result

1. The variables are now visible through the module-group variables in the data binding pad…
2. …and can be bound to data sources and parameters in test containers just like other variables.

Run a data-driven test

Now that we've finished our data-driven test, we can run it. This works the same as for other tests.

Screencast

The screencast “running a data driven test” walks you through the information found in this chapter.:

Watch the screencast now

Download the sample solution

This is the completed sample solution with all the instructions of the previous chapters carried out and ready to run.

Sample Data Driven Testing Complete

Install the sample solution:

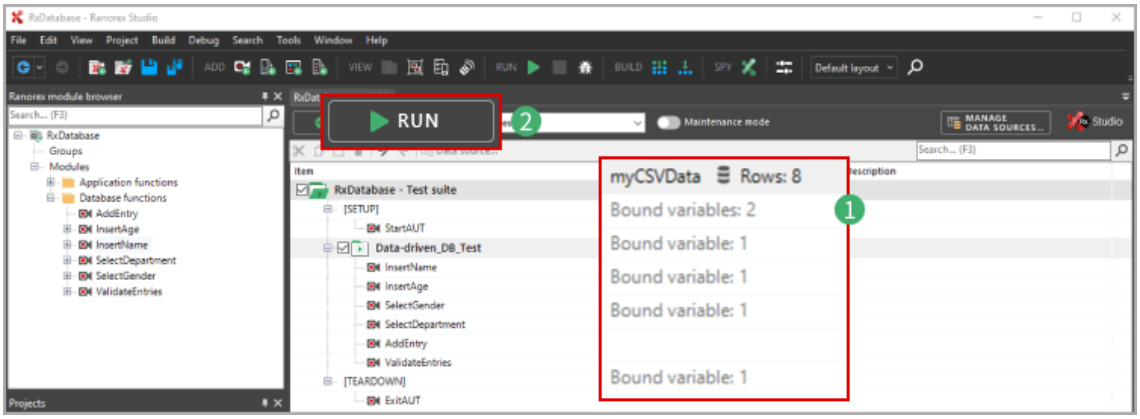
1. **Unzip** to any folder on your computer.
2. **Start** Ranorex Studio and **open** the solution file `RxDatabase.rxsln`

Hint

The sample solution is available for Ranorex versions 8.0 or higher. You must agree to the automatic solution upgrade for versions 8.2 and higher.

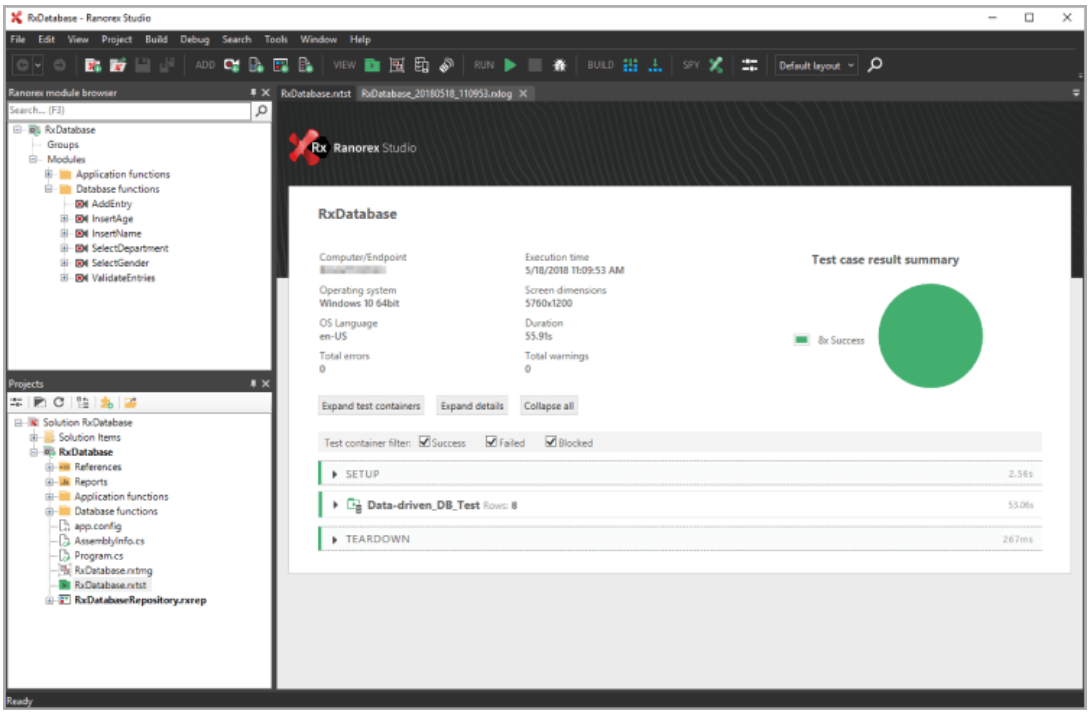
Run the test

1. **Check** if all variables are bound correctly.
2. **Click RUN**.



Result

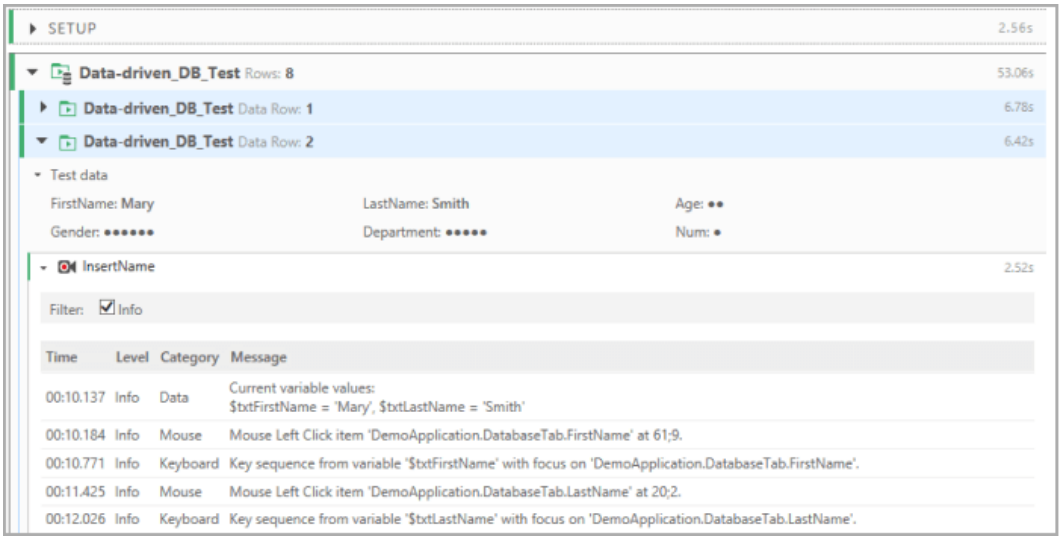
Once the test has finished, the report appears. The rows of the data-driven test case are shown in the results.



Result details for a data-driven test

When we take a more detailed look at the test case, we see that it was iterated once for each data row. So, in total, the test case was run 8 times.

The details for each iteration also show the current variable values for this iteration.



Further Reading

Reporting is explained in detail in Ranorex Studio fundamentals > ⇢ Reporting.

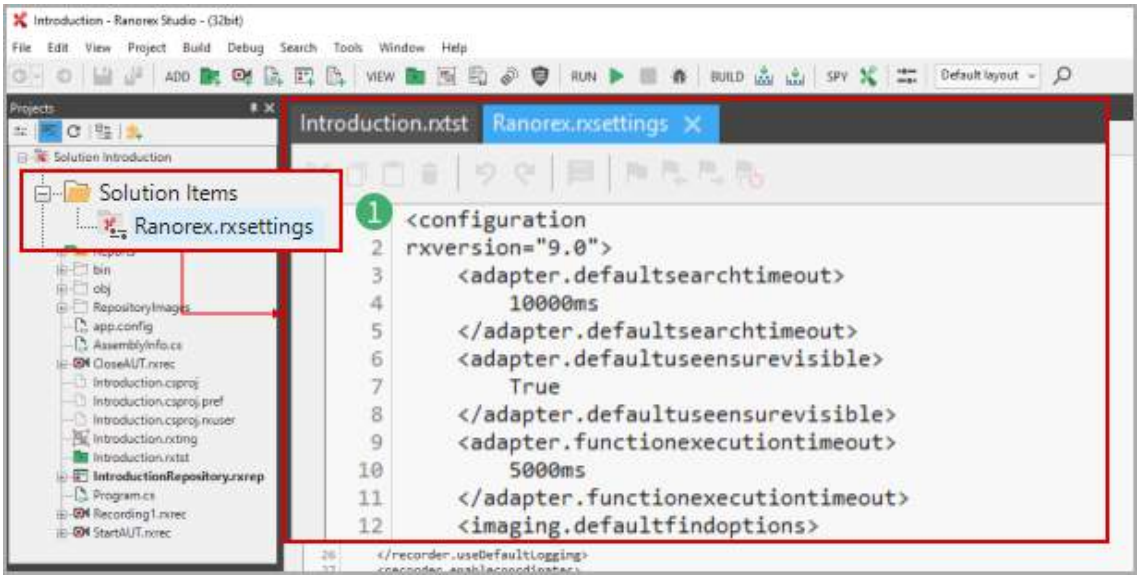
Microsoft Excel-free test execution on runtime machines

If you want to execute a data-driven test that uses Excel connectors in a ⇢ runtime or remote environment or in the Test Suite Runner, the target machine doesn't need to have an Excel license installed. Simply install the Microsoft Access Database Engine Redistributable >=2013 on the target machine and you can run the test.

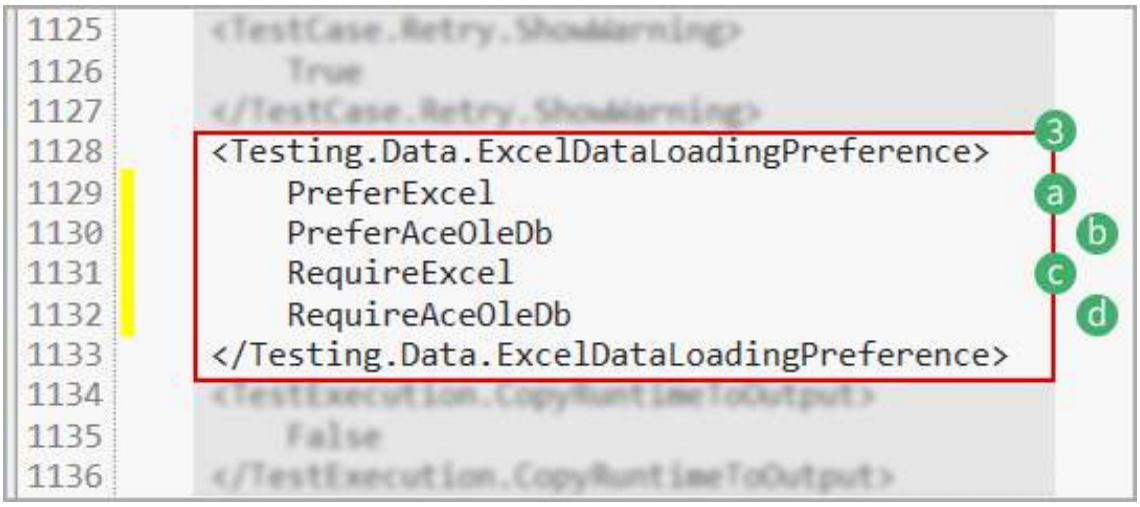
You can also view Excel data sources, but you have to select Auto-range or specify a range manually when creating the respective Excel data connector. Editing Excel data sources still requires MS Excel.

You can configure the behavior of the Excel-free execution implementation in the solution settings file.

1. In the projects view, **open** the solution settings file.



2. **Locate** the setting **<Testing.Data.ExcelDataLoadingPreference>.**
3. Enter one of four options (all are entered in the image for illustration purposes):
   a. **PreferExcel**: Default. Uses Excel if available, else uses AceOleDb (the free database engine). Test fails if neither is available.
   b. **PreferAceOleDb**: Uses AceOleDb if available, else uses Excel. Test fails if neither is available.
   c. **RequireExcel**: Uses Excel only. Test fails if it isn't available.
   d. **RequireAceOleDb**: Uses AceOleDb only. Test fails if it isn't available.



Parameters

In this chapter, you'll learn what parameters are, how to add them, and what you can use them for.

Parameters can be considered a data source with only one row, i.e. the simplest kind of data source. They are useful for keeping modules reusable and tests maintainable. They also allow you to pass variable values from one recording module to another.

> **Screencast**
>
> The screencast "Parameters" walks you through the information found in this chapter.:
>
> Watch the screencast now

Add a parameter

Parameters are added in the test suite view in an item's **Properties** dialog. You can add parameters to:

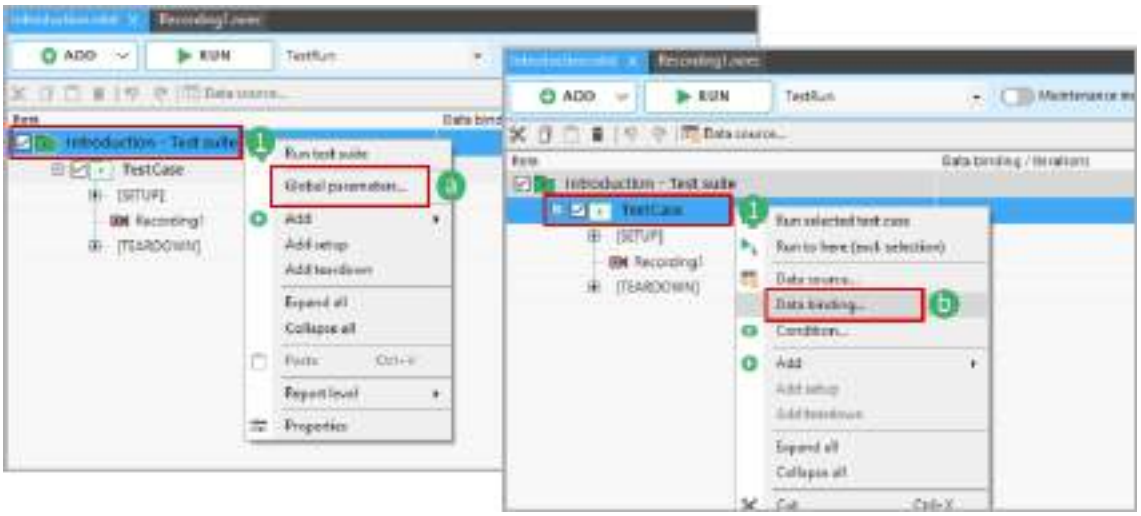
- The test suite item (= global parameters)
- Test containers (= local parameters)

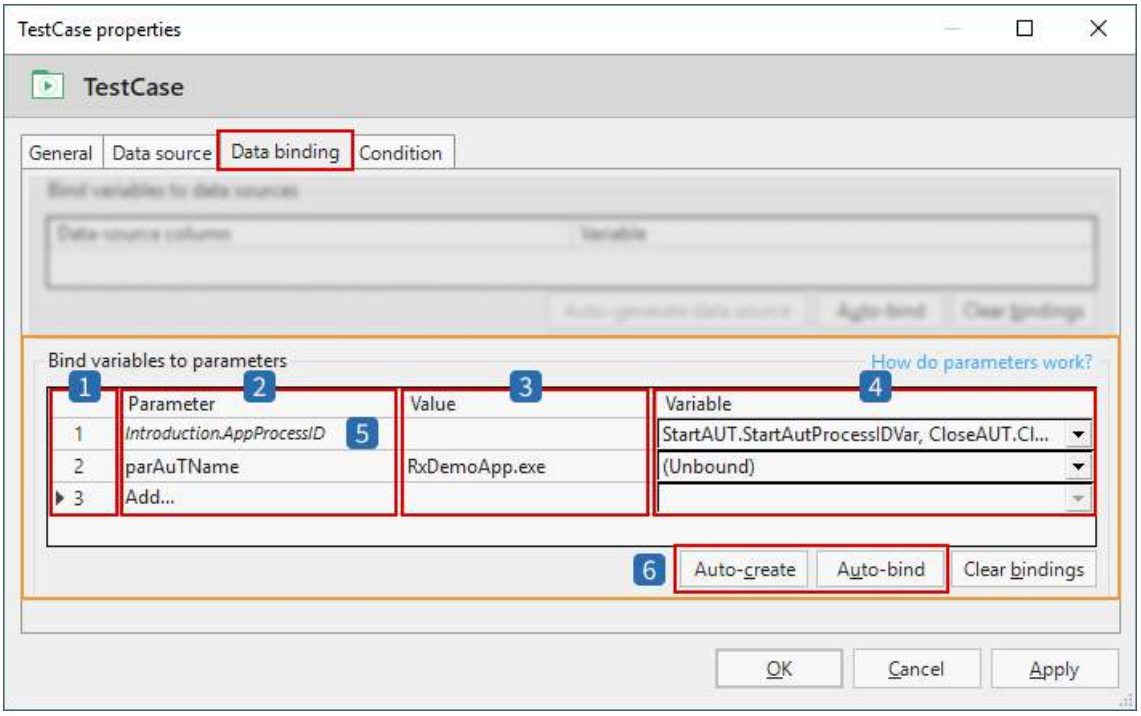
> **Note**
>
> When you add a parameter to an item, all its descendants inherit it.
>
> - For the test suite item, this means all test containers inherit it.
> - For test containers, this means that descendant test containers inherit it, but not siblings or parents.

The parameters dialog

To access the dialog for adding parameters:

1. **Right-click** the item you want to add a parameter to and…
   a. …for a test suite, **click Global parameters…**.
   b. …for a test container, **click Data-binding…**.





1. The number of parameters you've added. There is no limit.
2. The name of each existing parameter. Click on **Add row…** to enter the name of a new parameter.
3. The default value of each parameter. Assign a value to the new parameter. Any string is possible.
4. The current binding for each parameter. Use the drop-down menu to bind the new parameter to one or more variables.
5. Parameters in italics have been inherited from an ancestor. They can be edited only in the ancestor.
6. Click **Auto-create** to automatically create a parameter for any unbound variable. The new parameters will have the same name as the unbound

variables. Then click **Auto-bind** to automatically bind the new parameters to the respective variables.

Use parameters

In this section, we'll show you two examples of how to use parameters.

The **first example** is simple and useful for many kinds of tests. It is about using parameters to **increase the reusability** of a recording module.

The **second example** is more complex and more limited in its applications, but still highly useful. It is about using parameters to **pass a value from one variable to another in a different recording module during test execution**. This can increase efficiency and reduce test maintenance.

Example 1: Increase module reusability

> **Note**
>
> This example is quite short and so does not come with a dedicated sample solution.

Consider a typical AUT that offers different functions. It presents these functions on different screens that users access through the UI.

For example, there are several tabs in the Ranorex Studio Demo Application, each offering a different functionality:

To test these screens, you need to create several different test cases, one for each screen. Why? Because you're testing a different functionality on each screen, so the test steps will

be different in each case. However, one of the first steps always stays the same: Opening the AUT and bringing up the screen. It always consists of these actions:

1. Open the AUT.
2. Click the tab for the screen you want to bring up.

This makes for a highly reusable module, and reusable modules increase efficiency. But **there's one problem**: The name of the tab to click is different in each test case.

The easiest and most efficient way to solve this is with variables and parameters.

Define a variable

In the recording module **StartAUT**, we will make the repository item linked to the Click action variable.

1. Define a mouse-click action on a tab of the demo application and **follow** the instructions for repository variables in ⇢ Define variables. **Name** the variable **$varTab**.

Define parameter and link variable

Now we'll define a parameter in the first test case (Introduction_testing) and bind the repository variable to it.

1. **Right-click** a test case and **click Data-binding….**
2. Under **Parameters**, **name** the parameter **Tab** and **assign** it the value **Introduction**.

> **Note**
>
> The parameter value is the attribute value in the RanoreXPath that identifies the UI element. For the Introduction tab, this would be [@accessiblename='**Introduction**']. For the Test database tab, it would be [@accessiblename='**Test database**'].
>
> For more information, go to Ranorex Studio advanced > ⇢ RanoreXPath

3. **Bind** it to the variable **varTab** and **click OK**.

4. **Repeat** this for the other test cases and **replace** the parameter value with the attribute value that identifies the respective tab, i.e. Test database, Image-based automation, etc.

And that's it. When you run the test, it will go through the tabs automatically.

Example 2: Pass values across modules during test execution

In this example, we'll show you how to take a value generated in one module during test execution and pass it to another module in the same test run. We'll accomplish this with variables, parameters, and the Get value action.

To explain this, we'll use a small database test in the Ranorex Studio Demo Application.

Download the sample solution

As this example is more complex, we've created sample solutions for you. Download the file below to follow along with the instructions. A finished sample is available at the end of this section.

Sample Parameter

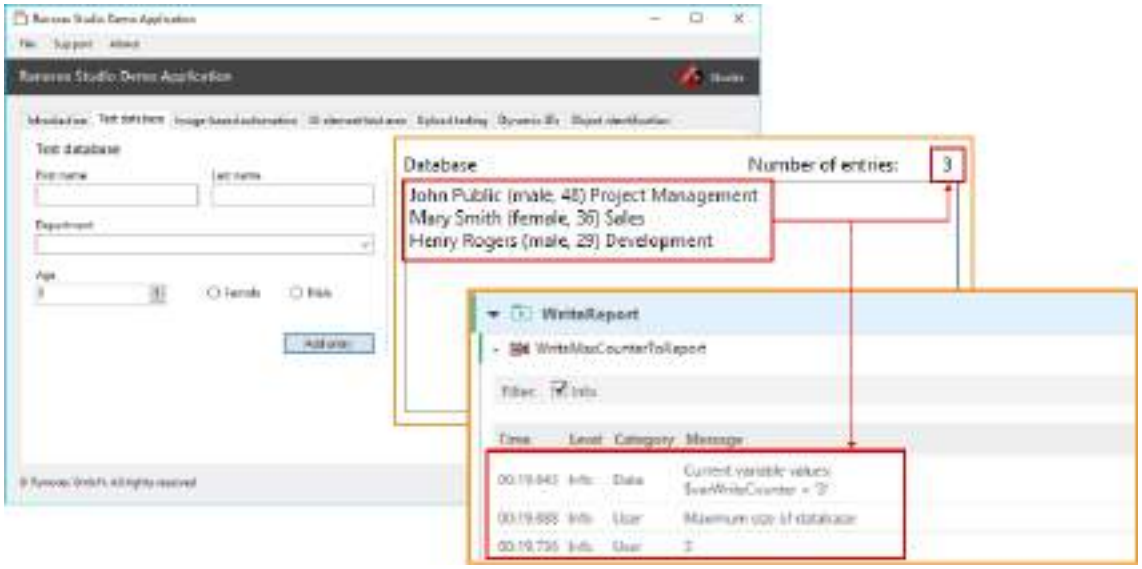
Install the sample solution:

1. **Unzip** to any folder on your computer.
2. **Start** Ranorex Studio and **open** the solution file.

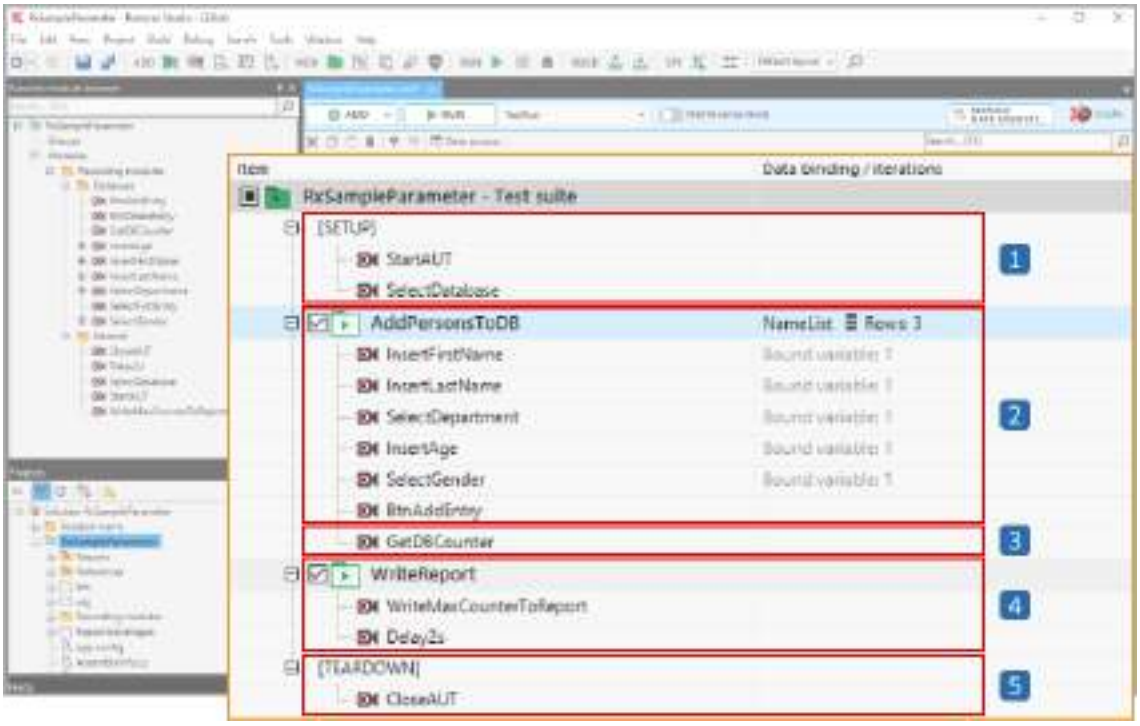
> **Hint**
>
> The sample solution is available for Ranorex versions 8.0 or higher. You must agree to the automatic solution upgrade for versions 8.2 and higher.

Test description

In our example test, we'll use data-driven testing to enter a number of persons into the database of the Ranorex Studio Demo Application. With each entry, a counter showing the number of entries will increase. At the end of the test run, we want to log the total number of entries to the report.



Initial test suite



1. Starts the Demo App and clicks the Database register.
2. The first six recording modules add the entries from a data source into the database, as explained in the previous subchapters.
3. The recording module GetDBCounter will extract the counter value and write it to a variable.
4. The recording module WriteMaxCounterToReport will log the counter value to the report.
5. Closes the AUT.

The counter value is **generated dynamically during test execution** depending on the number of entries added. Depending on changes to the data source or the inclusion of conditions, for example, the final number will vary. Therefore, **we can't reliably know** what the counter value will be during test execution, or it would require too much effort to find out each time. To log the correct value to the report automatically, we will need to make use of **variables and parameters**.

Extract value and write it to a new variable

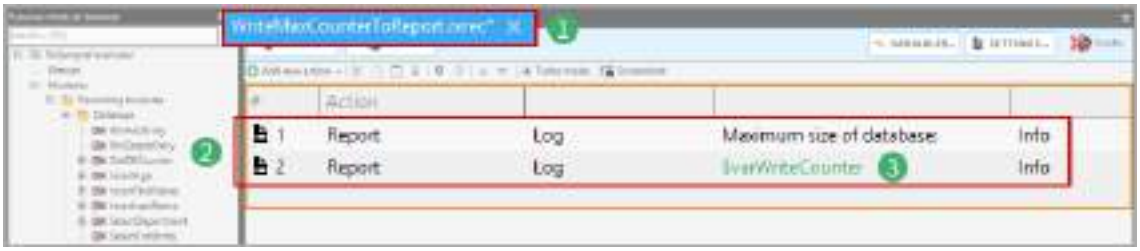
First, we'll use a Get value action to extract the counter value after all entries have been added during test execution and write it to a new variable:

1. **Open** the recording module **GetDBCounter**.
2. **Add** a Get value action that is linked to the repository item for the entry counter.
3. **Create** the new variable **$varCounter**. The value of the counter will be written to this variable.

Log counter value to report

Now, we'll add the action to log the counter value to the report.

1. **Open the recording module WriteMaxCounterToReport.**
2. **Add** two new Log message actions as in the image below.
3. Add the new variable $varWriteCounter. This variable will receive the value from the GetDBCounter module and log it to the report.



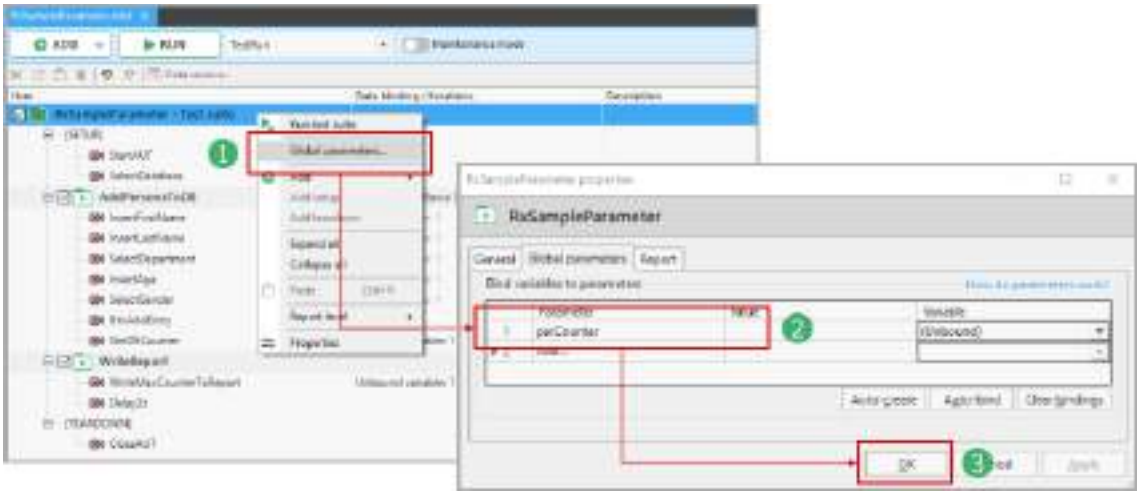
Create a transfer parameter

During the test run, the variable in **GetDBCounter** will now receive the dynamically generated counter value. However, the variable in **WriteMaxCounterToReport** can't retrieve the value yet, because variables and their values are local to their modules. We need to "link" the variables. We do this by binding them all to the same parameter.

The parameter that links the variables needs to be located in a **common ancestor of both modules**. In our case, the modules are in two separate test containers, so the first common

ancestor is the test suite item RxSampleParameter. If the two modules were in the same test container, you could add the parameter there.

1. **Right-click** the test suite item **RxSampleParameter** and **click Global parameters…**.
2. Under **Parameters**, **add** a parameter called **parCounter.**
3. **Click OK.**

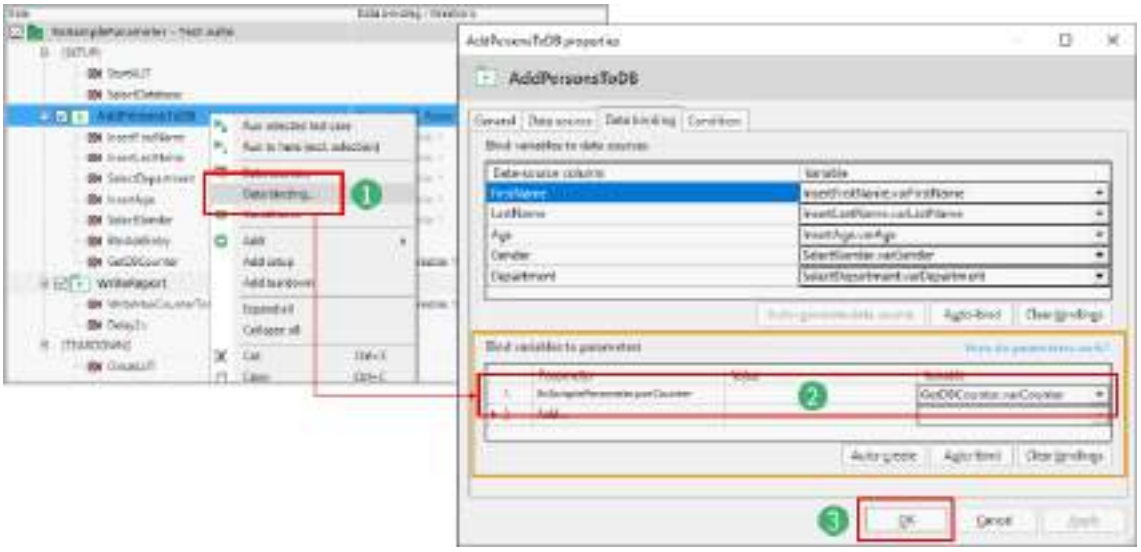


> **Note**
>
> We leave the value empty to show that this is a parameter that receives its value during test execution. You can enter anything; the value will be replaced for the test run.

Link variables to parameter

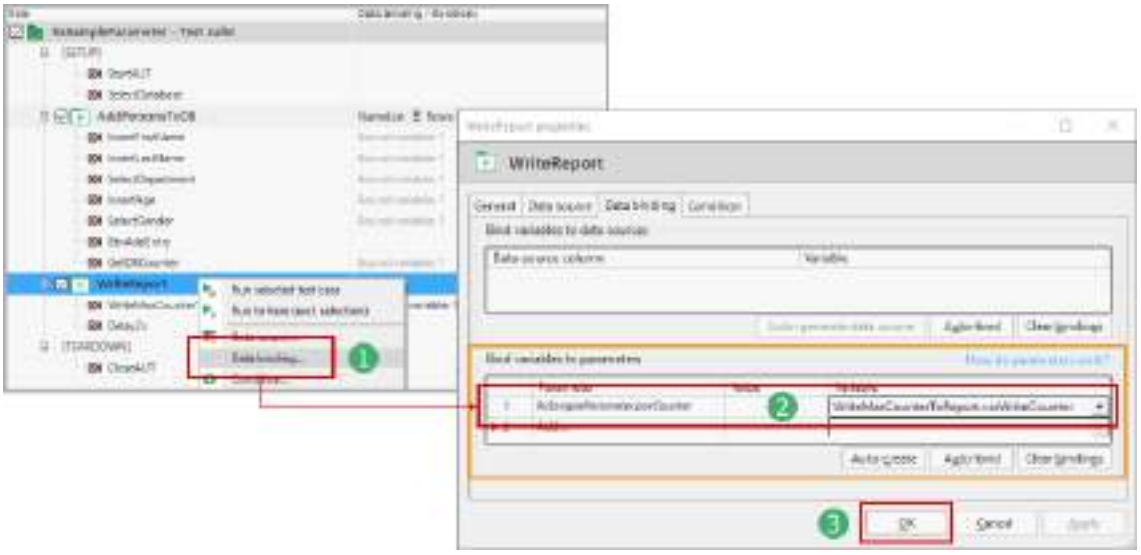
With the parameter added, we can now bind the variables to it and link them together. This way, the dynamically generated value will be passed from one module to another.

1. **Right-click** the test case **AddPersonToDB** and **click Data binding…**.
2. Under **Parameters**, bind the inherited parameter in italics to the variable **$varCounter**.
3. **Click OK**.



Repeat the process for the second variable in the other test case:

1. **Right-click** the test case **WriteReport** and **click Data binding…**.
2. Under **Parameters**, bind the inherited parameter in italics to the variable **$varWriteCounter**.
3. **Click OK**.



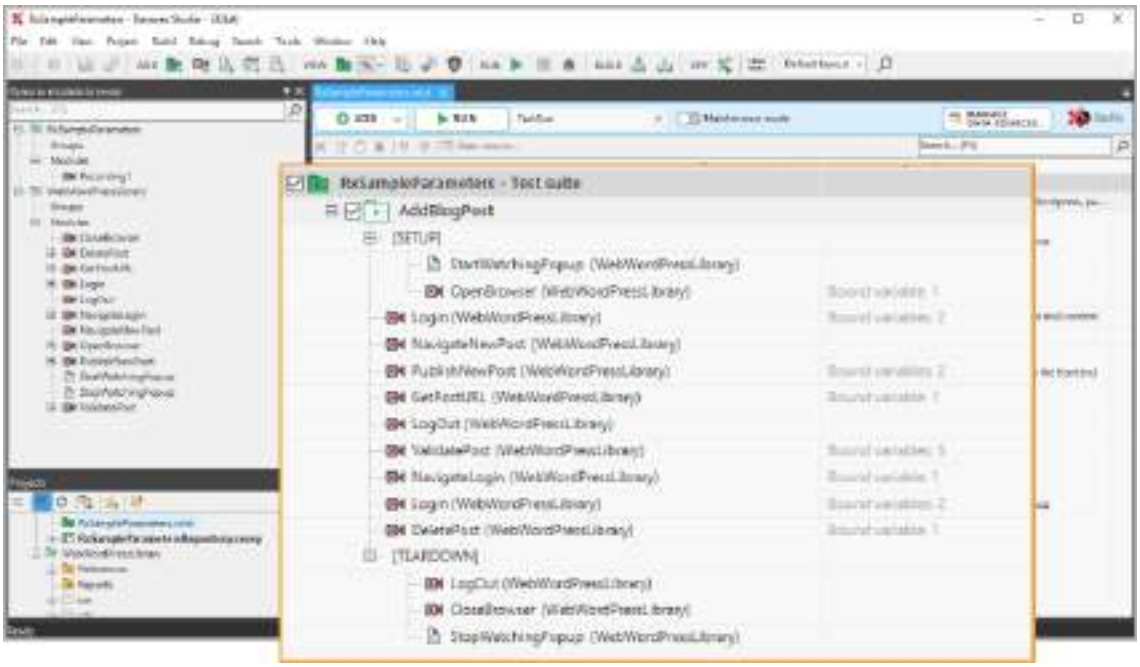
In this way, you can pass values from one module to another or others in any test container, so long as **all modules have access to the same parameters**. Naturally, this also only works **chronologically**, i.e., the giving module needs to be executed before the receiving module.

After you've run the test, you will see the counter value is logged to the report correctly:

Finished sample solution

Web test example

Extracting and passing on dynamically generated values is also relevant in web testing. The web test sample included in Ranorex Studio (Ranorex Studio start page > Sample solutions > Web example) demonstrates this:



The module **PublishNewPost** enters a post title and content and then clicks the Publish button in WordPress. When you publish a post in WordPress, it **automatically generates a unique URL** for the published post.

This URL is needed in the **ValidatePost** module, so the test can navigate to the blog post and see if it was published correctly. It's also needed in the **DeletePost** module to navigate to the correct post to be deleted.

However, as with the counter value in the Demo App, **we don't know what the URL will be** during the test run.

Therefore, the module **GetPostURL** extracts the value, writes it to a variable, and passes it on to the other two modules. This works in the same way as explained in the previous example, except here, all modules are contained in the same test case. Therefore, the linking parameter is located in this test case and all variables are bound to it.

Conditions and rules

Conditions and rules give you additional control over test execution. As the name suggests, they execute tests based on whether a certain condition is met. Conditional execution depends on values from a data source or a parameter, which is why this topic is explained as part of data-driven testing.

In this chapter, we'll show you how to set up a condition with a simple rule to create a conditional test case. We'll explain the related settings along the way. You can download the completed example at the end of this chapter.

Screencast

The screencast "Conditions" walks you through information found in this chapter.:

Watch the screencast now

Download the sample solution

To follow along with the instructions in this chapter, download the sample solution file from the link below:

Sample Data Driven Condition Start

Note

We've disabled two modules in this solution because they would interfere with the condition (validation) or prevent you from seeing the results in the Ranorex Studio Demo Application (teardown).

Install the sample solution:

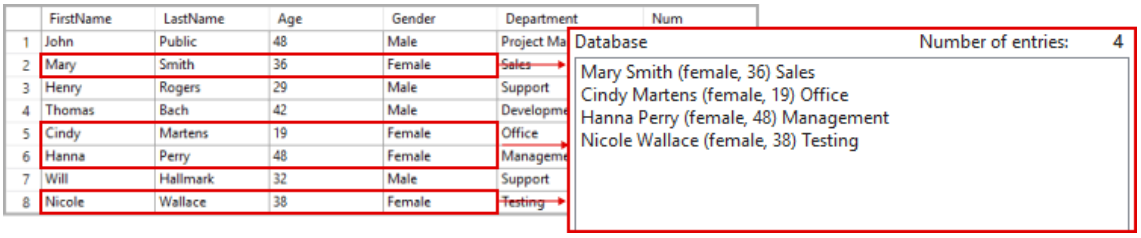
1. **Unzip** to any folder on your computer.
2. **Start** Ranorex Studio and **open** the solution file `RxDatabase.rxsln`

Hint

The sample solution is available for Ranorex versions 8.0 or higher. You must agree to the automatic solution upgrade for versions 8.2 and higher.

Test scenario

For our example, we'll use the data-driven test of the previous chapters. Only this time, we want to insert just the female staff members into the database of the Ranorex Studio Demo Application.



In summary:

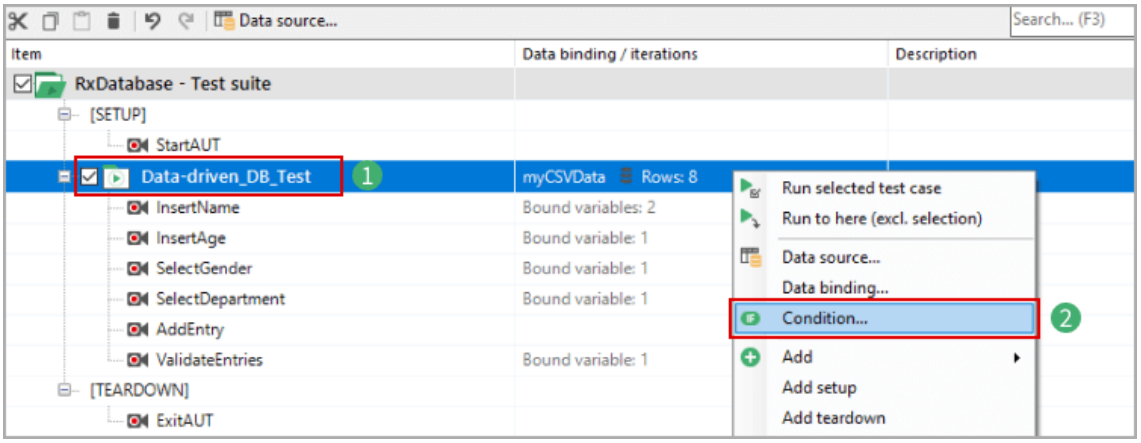
- Normally, 8 staff members are inserted into the database.
- We want to modify the test case so that it **enters only female staff members** into the database.
- We want to accomplish this with a **condition**.

Add a condition

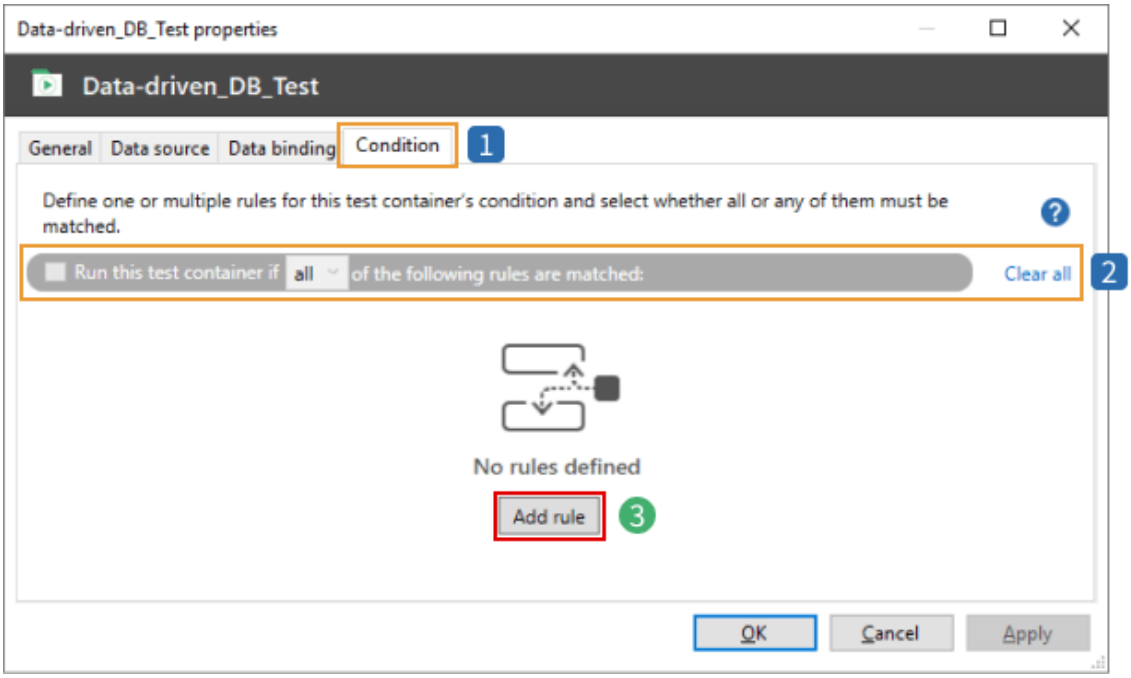
- You can add a condition to test cases or smart folders only.
- Only one condition per test container.
- Conditions can contain and are defined by one or more rules.
- The entire process is the same for both types of test containers.

To add a condition:

1. **Right-click** the smart folder or test case you want to add the condition to.
2. **Click Condition…**



3. **Click Add rule** to define a rule for this condition.

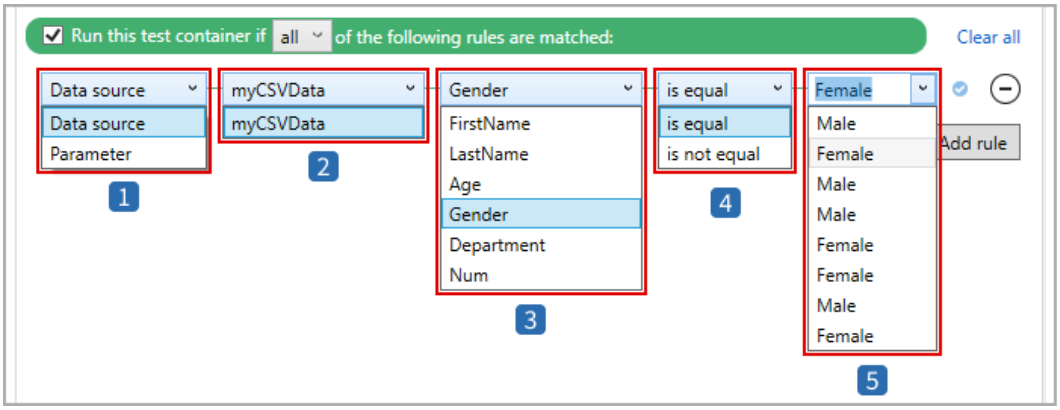


1. The **Condition** tab of the test container properties dialog.
2. The inactive condition, with no rules defined below it yet.

Define rules

Conditions are defined by rules. You can add up to 10 rules to a condition. Rules always follow the same pattern and check whether a data source or parameter value is equal or not equal to a control value.

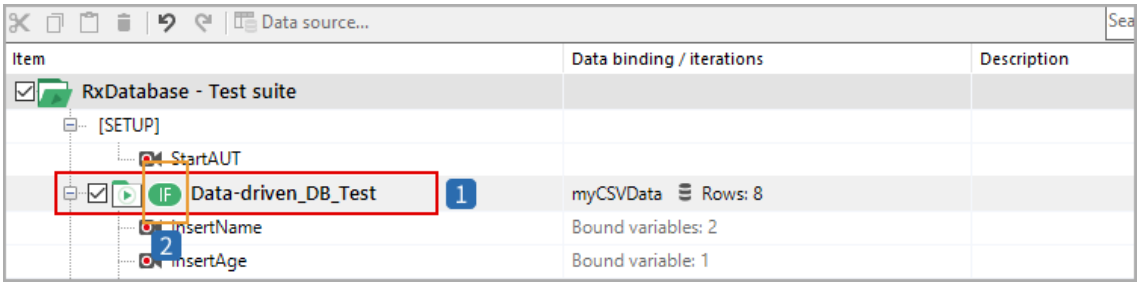
1. **Define** a rule from left to right by selecting the required details and **click OK** when you're done.



1. **Data source/Parameter**: Select whether the value you want to check is in a data source or a parameter.
2. **Specific data source/parameter**: Select the specific data source or whether it is a local/global parameter that contains the value.
3. **Column/row**: Select the column (data sources) or the row (parameters) that contains the value.
4. **Operator**: Select whether the value you want to check must be equal or not equal to the control value.
5. **Control value**: Select a possible control value from the data source/parameter.

Result

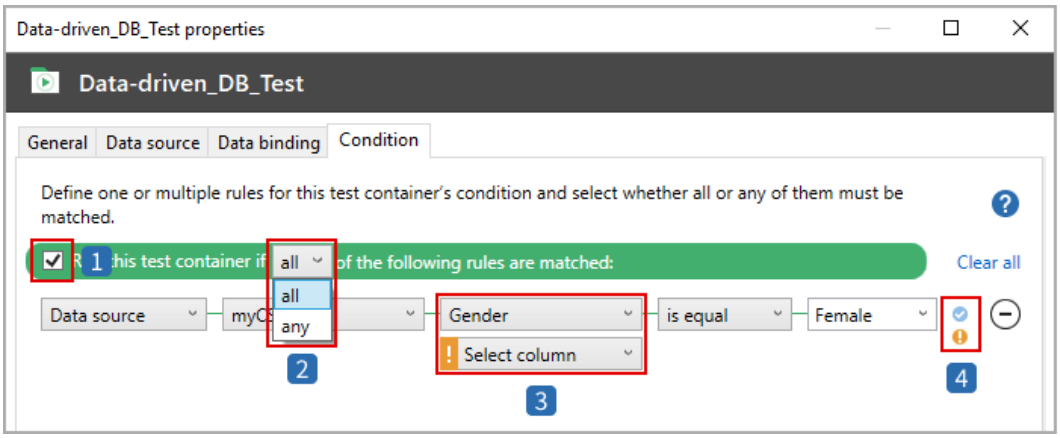
If you defined the rule just as above, you now have a condition with one complete rule. In the test suite view, this will be indicated as follows:



1. Conditional test case.
2. Indicator showing that the test case has an active, defined condition.

Condition settings

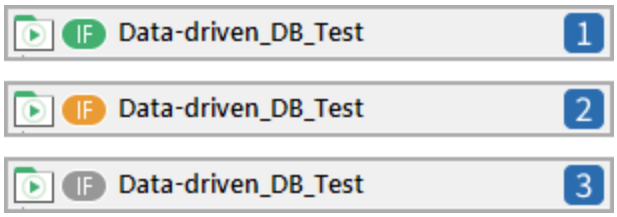
The condition dialog has various settings in addition to the ones we've already used:



1. **Activation checkbox**: Quickly switch between active/inactive condition. Does not delete any defined rules.
2. **Condition operator**: Sets when the condition is met and the test container is run.
3. **Incomplete rule indicator**: Appears wherever a rule is missing information.
4. **Rule completeness indicator**: Shows whether the rule is complete or incomplete.

Condition status

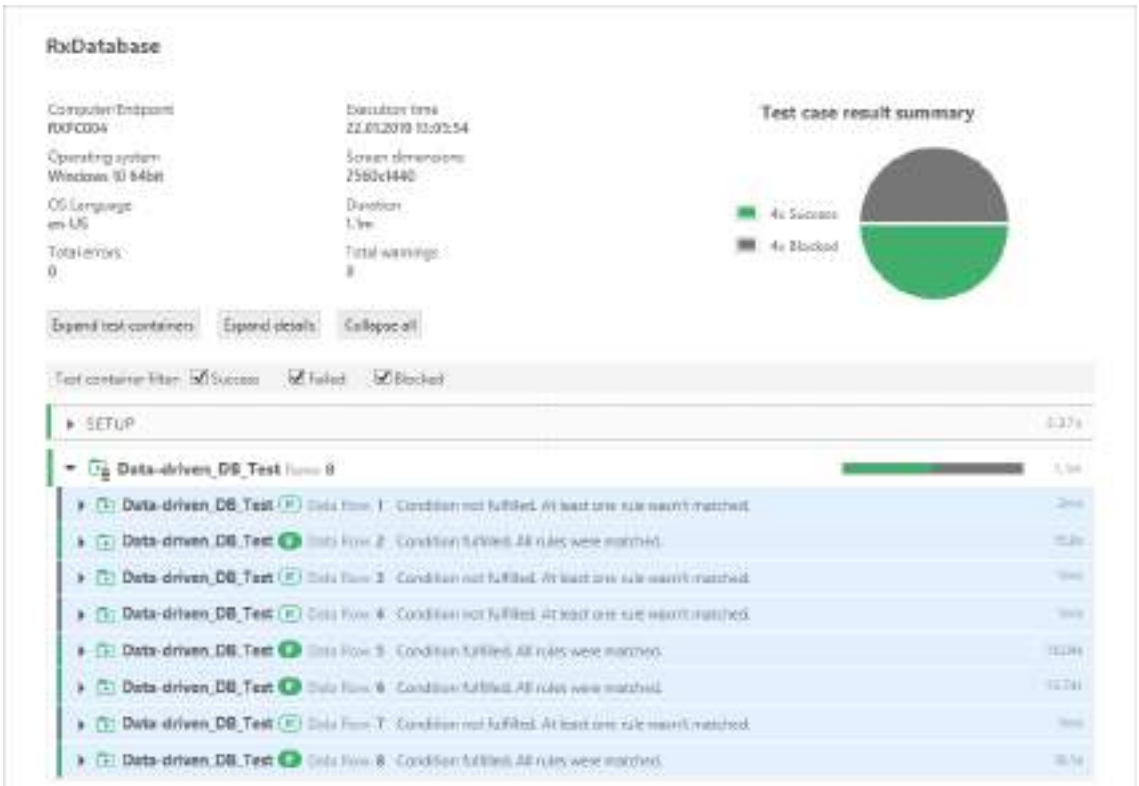
Depending on their completeness and activation status, conditions are indicated by three different symbols in the test suite view:



1. Active condition with one or more complete rules.
2. Active condition with one or more incomplete rules.
3. Inactive condition with one or more complete or incomplete rules.

Run the test

We've completed the condition and are ready to run the test. Run the test from the test suite view as usual. You will notice the database in the Ranorex Studio Demo Application only contains female staff members. The report will also reflect the condition, with iterations where the condition wasn't met marked as blocked:



Download the completed sample solution

You can download the completed sample solution with all the steps in this chapter carried out and ready to run:

Install the sample solution:

1. **Unzip** to any folder on your computer.
2. **Start** Ranorex Studio and **open** the solution file `RxDatabase.rxsln`

> **Hint**
>
> The sample solution is available for Ranorex versions 8.0 or higher. You must agree to the automatic solution upgrade for versions 8.2 and higher.

Which type of variable are you looking for?

> **Reference**
>
> If you're looking for an explanation on how to define and use **action, validation, or repository variables using the Ranorex Studio UI**, go to Ranorex Studio advanced > Data-driven testing > ⇢ Define variables

> **Reference**
>
> If you're looking for an explanation on how to define and use **module variables in code modules**, go to Ranorex Studio expert > Code modules > ⇢ Module variables and data-driven testing

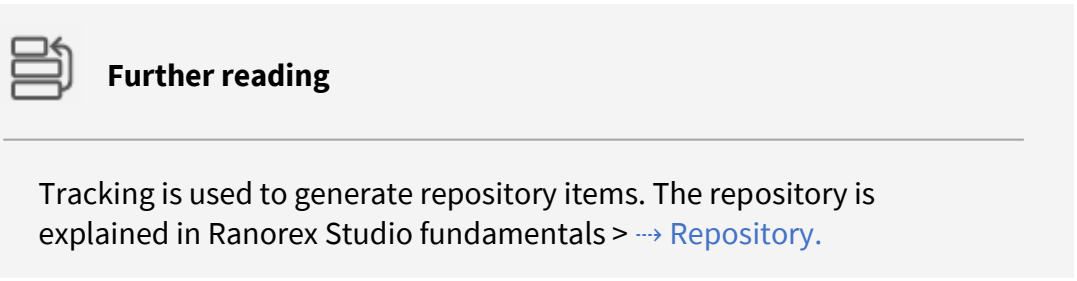
Tracking UI elements

Automated UI testing is based on identifying UI elements and generating a representation of them that the automation tool can work with, i.e. repository items with a unique RanoreXPath. In Ranorex Studio, this is accomplished through tracking UI elements.

In this chapter, we'll explain the different ways you can track UI elements in Ranorex Studio.

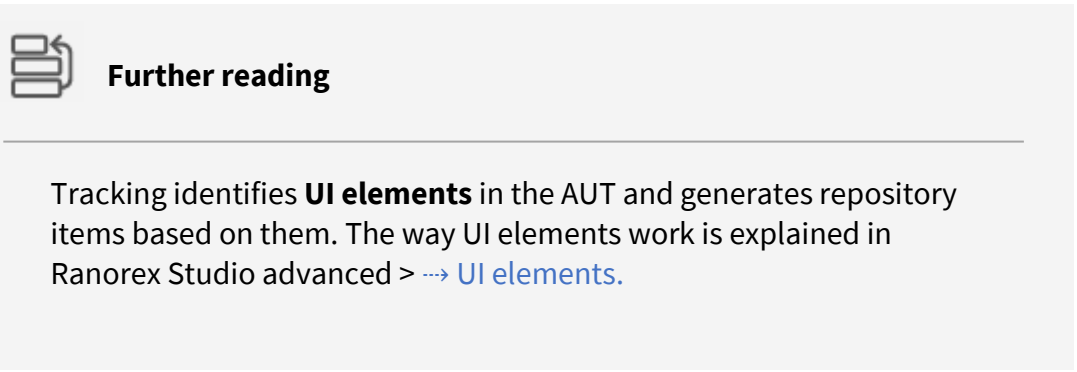
Further reading

Tracking is closely connected to the concepts of UI elements, repository items, RanoreXPath, and the Ranorex Spy tool.



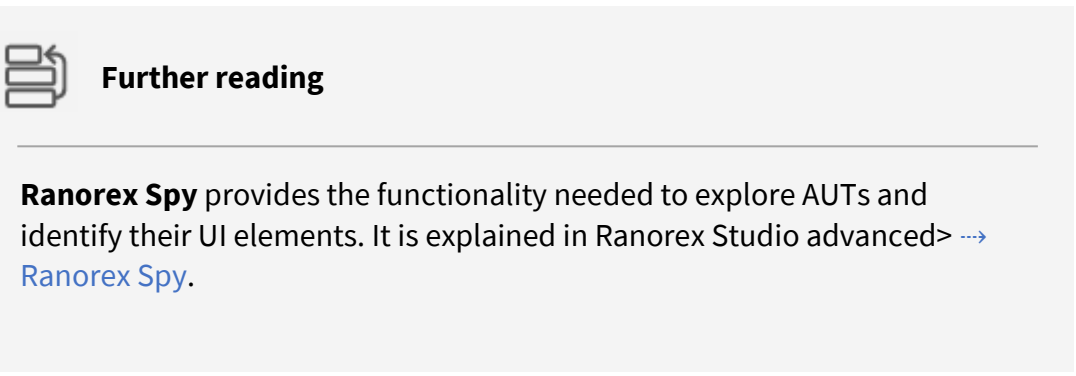
Further reading

Tracking is used to generate repository items. The repository is explained in Ranorex Studio fundamentals > ⇢ Repository.



Further reading

Tracking identifies **UI elements** in the AUT and generates repository items based on them. The way UI elements work is explained in Ranorex Studio advanced > ⇢ UI elements.



Further reading

Ranorex Spy provides the functionality needed to explore AUTs and identify their UI elements. It is explained in Ranorex Studio advanced> ⇢ Ranorex Spy.

Further reading

Ranorex Studio uses the **RanoreXPath** to identify UI elements represented by repository items. This concept is explained in Ranorex Studio advanced > ⇢ RanoreXPath.

Track by recording

When you record test steps in Ranorex Studio, UI elements are automatically tracked and therefore identified as you perform actions on them. This is tracking by recording. If you've read through the chapters in Ranorex Studio fundamentals, you should already be familiar with this method. We'll take a quick look at it again on this page by way of a simple example.

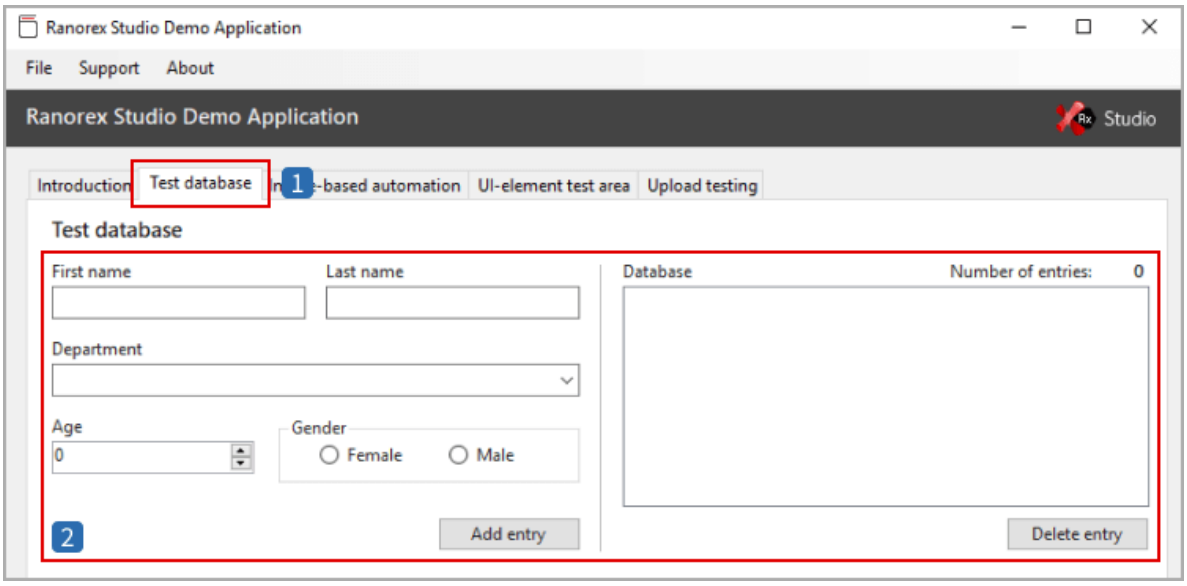
Screencast

The screencast "track by recording" walks you through the information found in this chapter.:

Watch the screencast now

Test example definition

To explain tracking by recording, we'll record a click on a UI element in the Test database of the Ranorex Studio Demo Application.

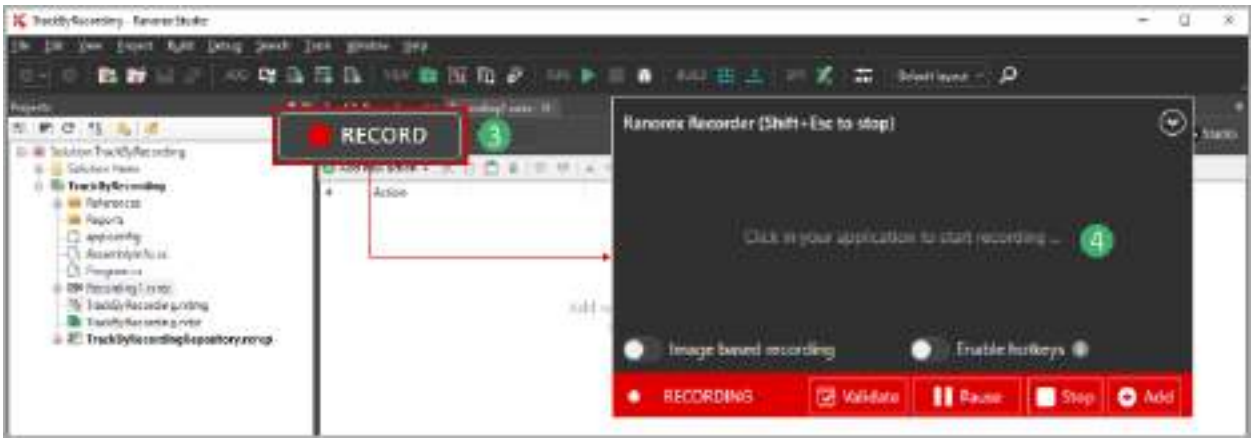


1. The Test database tab in the demo application.
2. The test database working environment.

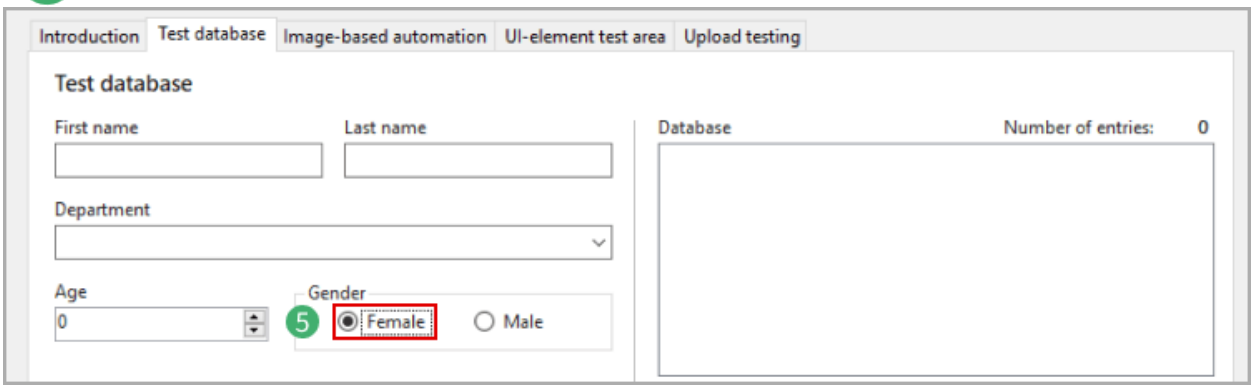
Track by recording

Now let's see how tracking by recording actually works.

1. **Start** the demo application and **click** the Test database tab.
2. **Start** Ranorex Studio and **create** a new blank solution.
3. **Open** the default recording module and **click RECORD**.
4. Ranorex Studio disappears and the Recorder control center appears, indicating an active recording.



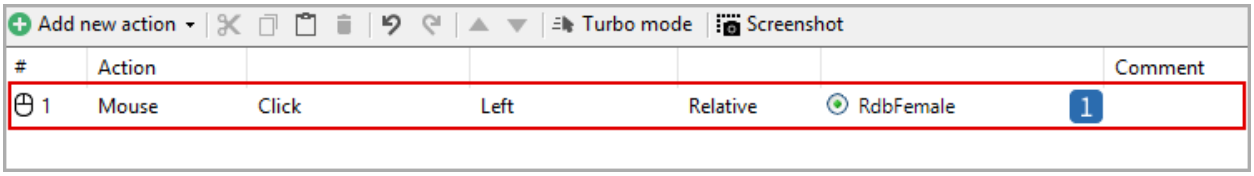
5 In the Test database tab, **click** the **Female** radio button.



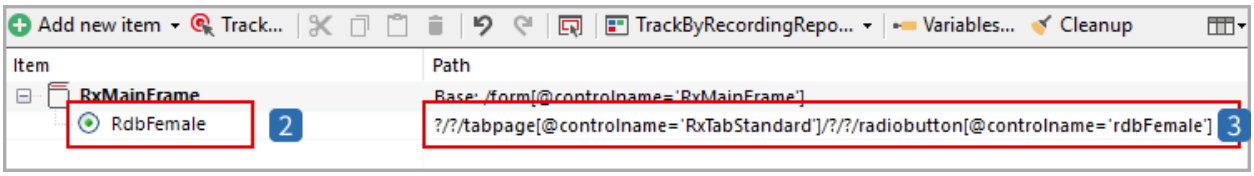
6 **Click Stop** in the Recorder control center.

Results

Once you've stopped recording, Ranorex Studio displays the recording module view with the actions table and the current repository.



1 Mouse click action linked to the repository item RdbFemale that represents the Female radio button.



2 The repository contains the repository item RdbFemale. This repository item represents the Female radio button and is the result of the tracking process.

3 The RanoreXPath of the repository item. This path identifies the position of the UI element in the AUT's UI.

Tracking mechanism for tracking by recording

- While recording, Ranorex Studio monitors user interactions with the UI and automatically tracks UI elements.
- When a user interaction occurs, Ranorex Studio identifies the targeted UI element and stores it as a repository item, which is a representation of the UI element.
- Usually, one repository item represents one UI element.

- Ranorex Studio recognizes when a UI element is used more than once and then reuses the corresponding repository item.

Track button

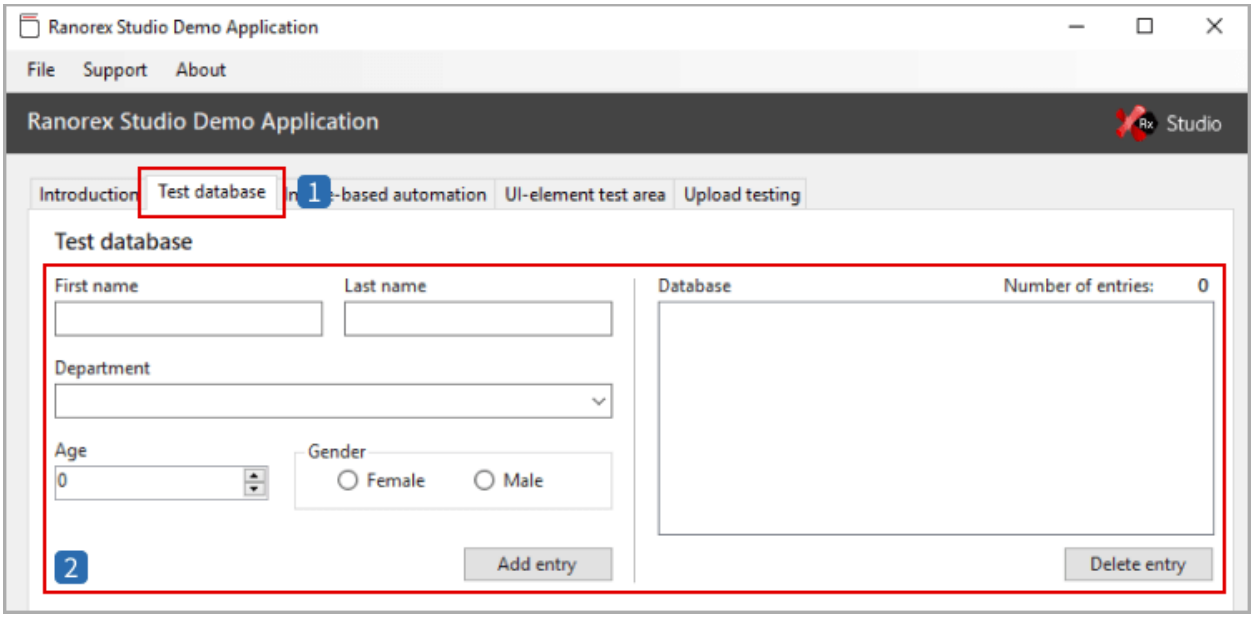
On this page, you'll find out how to track and identify single UI elements with the Track button from a recording module or repository file in Ranorex Studio.

Screencast
The screencast "track button" walks you through the information found in this chapter.:

Watch the screencast now

Test example definition

To explain tracking with the Track button, we'll use the Test database of the Ranorex Studio Demo Application.



1 The Test database tab in the demo application.

2 The test database working environment.

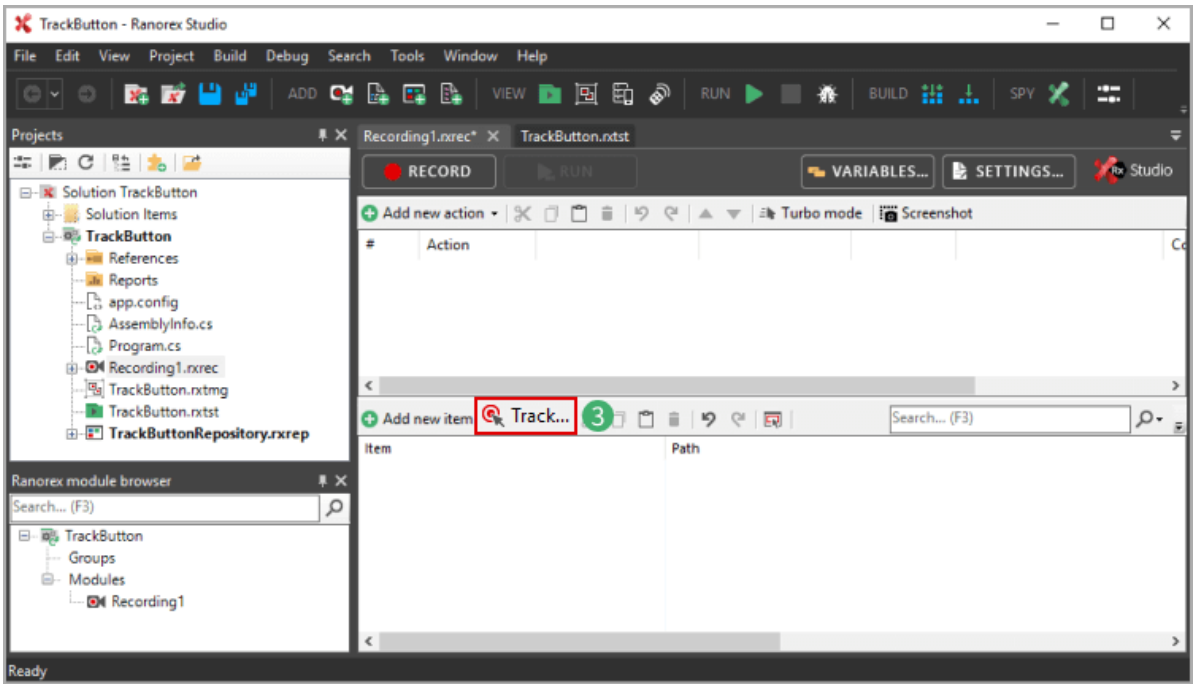
Track button

The Track button is available in different locations:

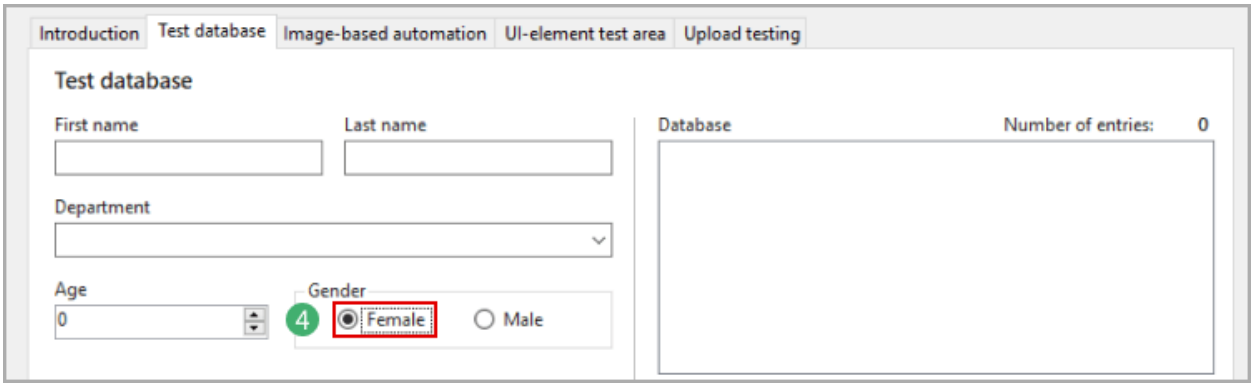
- In an opened recording module in the repository toolbar
- In an opened repository file
- In Ranorex Spy

> **Note**
>
> The first two Track buttons work the same. You'll likely use the first one more frequently, which is why we'll focus on it on this page.
>
> The Track button in Spy works a little differently, which is why we explain it separately as part of the Ranorex Spy chapter.

1. **Start** the demo application and **click the Test database** tab.
2. **Start** Ranorex Studio and **create** a new blank solution.
3. **Open** the default recording module and **click** the **Track…** button in the repository toolbar.



4 Ranorex Studio is minimized. In the demo application, **mouse over** the **Female** radio button and **click** it when the red frame surrounds it as below to track it.

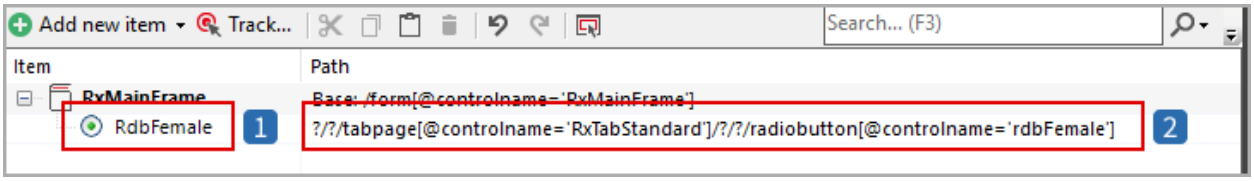


Note

You can track only one element this way. After tracking this element, you're returned to Ranorex Studio. To track another element, you need to repeat the process.

Results

Once the UI element has been tracked and identified, the tracking process ends and you're returned to Ranorex Studio. There, you can see the created repository item of the UI element.

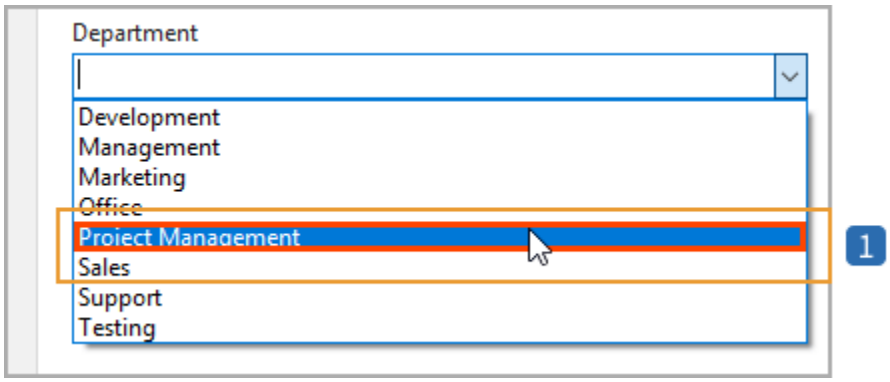


1 The repository contains the repository item RdbFemale. This repository item represents the Female radio button and is the result of the tracking process.

2 The RanoreXPath of the repository item. This path identifies the position of the UI element in the AUT's UI.

Tracking mechanism for the Track button

- When you click the Track button, the tracking process starts and is ready to track and identify exactly one UI element.
- The tracking process ends automatically after tracking one UI element.
- Pressing F12 during the tracking process **pauses** tracking. This allows you to click elements without tracking them. This way, you can track UI elements hidden in drop-down lists or menus which need to be opened first.



1 Pressing F12 during the tracking process allows you to track the menu item **Project Management** from a drop-down list in the demo application.

Instant tracking

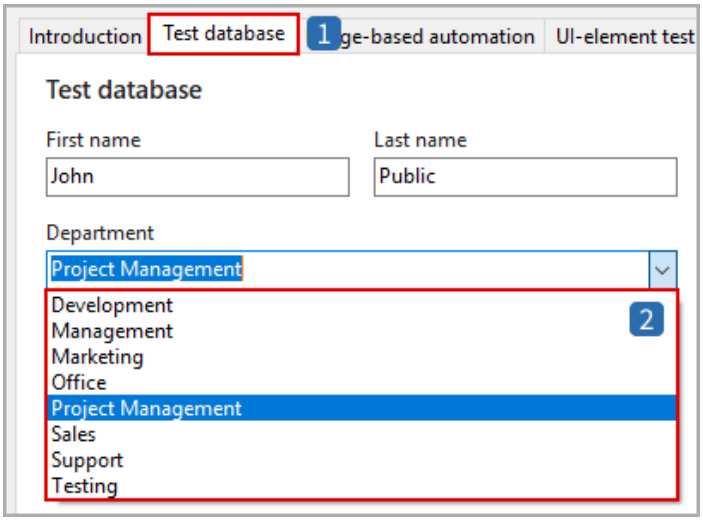
Instant tracking works by pressing Ctrl + WIN. This way of tracking is particularly useful to quickly add multiple repository items from Ranorex Studio or Spy without repeatedly using the Track button. On this page, you'll find out how it works.

Screencast
The screencast "instant tracking" walks you through the information found in this chapter.:

Watch the screencast now

Test example definition

To explain tracking with the Track button, we'll use the Test database of the Ranorex Studio Demo Application. The UI element we'll track is one of the list elements in the Department drop-down menu.



1. The Test database tab in the demo application.
2. The test database working environment with the Department drop-down menu.

Instant tracking

1. **Start** Ranorex Studio and **create** a new blank solution.
2. **Start** the demo application and **click the Test database** tab.
3. **Click** the **Department** drop down menu.
4. **Mouse over** the list element you want to track.
5. **Press** Ctrl + WIN to track the list element.

Tracking mechanism

- You can use instant tracking if Ranorex Spy is started or if you have a recording module or repository file opened in Ranorex Studio.
- You can instant track as many UI elements as you want in one go. You don't need to switch back to Ranorex Studio or Spy.

Ranorex Spy

Ranorex Spy is the component of Ranorex Studio that makes it possible to explore and analyze the UI of desktop, mobile, and web applications for the purpose of identifying UI elements. Spy captures all running applications (according to your >⇢ whitelist) and displays them and their subelements in a tree view. It therefore recognizes UI elements, identifies them, assigns a RanoreXPath to them, and ultimately makes them available as repository items to Ranorex Studio.

Spy is available as a standalone version and from within Ranorex Studio.

On this page, you'll find out how to start Spy and use its basic functions.

Related chapters

Ranorex Spy is an advanced topic that is closely connected to the concepts of RanoreXPath and UI elements. Together, they explain how UI element identification in Ranorex Studio works. We therefore recommend you also study these chapters.

Further reading

RanoreXPath is explained in Ranorex Studio advanced > ⇢ RanoreXPath.

Further reading

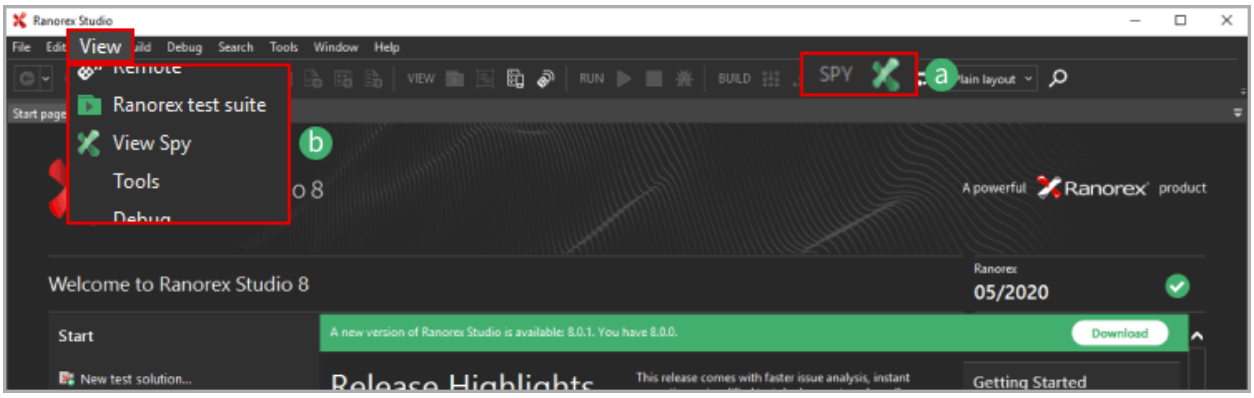
UI elements **are explained in Ranorex Studio advanced > ⇢ UI elements.**

Start Ranorex Spy

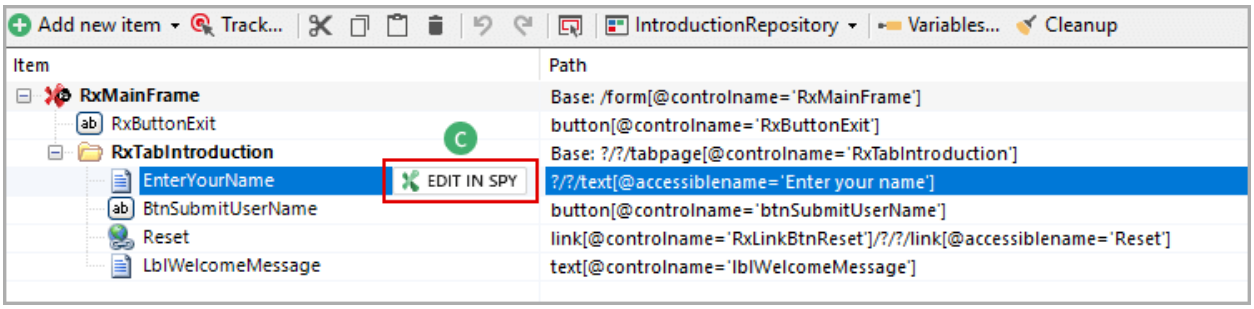
Spy is available integrated into Ranorex Studio and as a standalone tool.

Integrated Spy

a. In the Ranorex Studio toolbar, **click** the Spy icon.
b. **Click View > View Spy.**

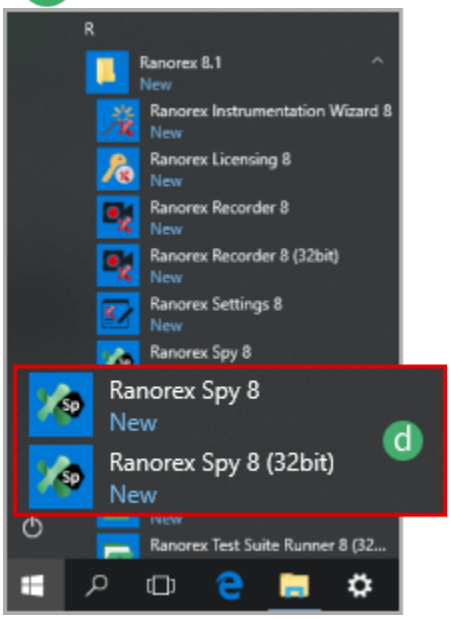


c In a repository, **click EDIT IN SPY** next to a selected repository item.



Standalone version

d **Open** Ranorex Spy from the Start Menu.



Hint

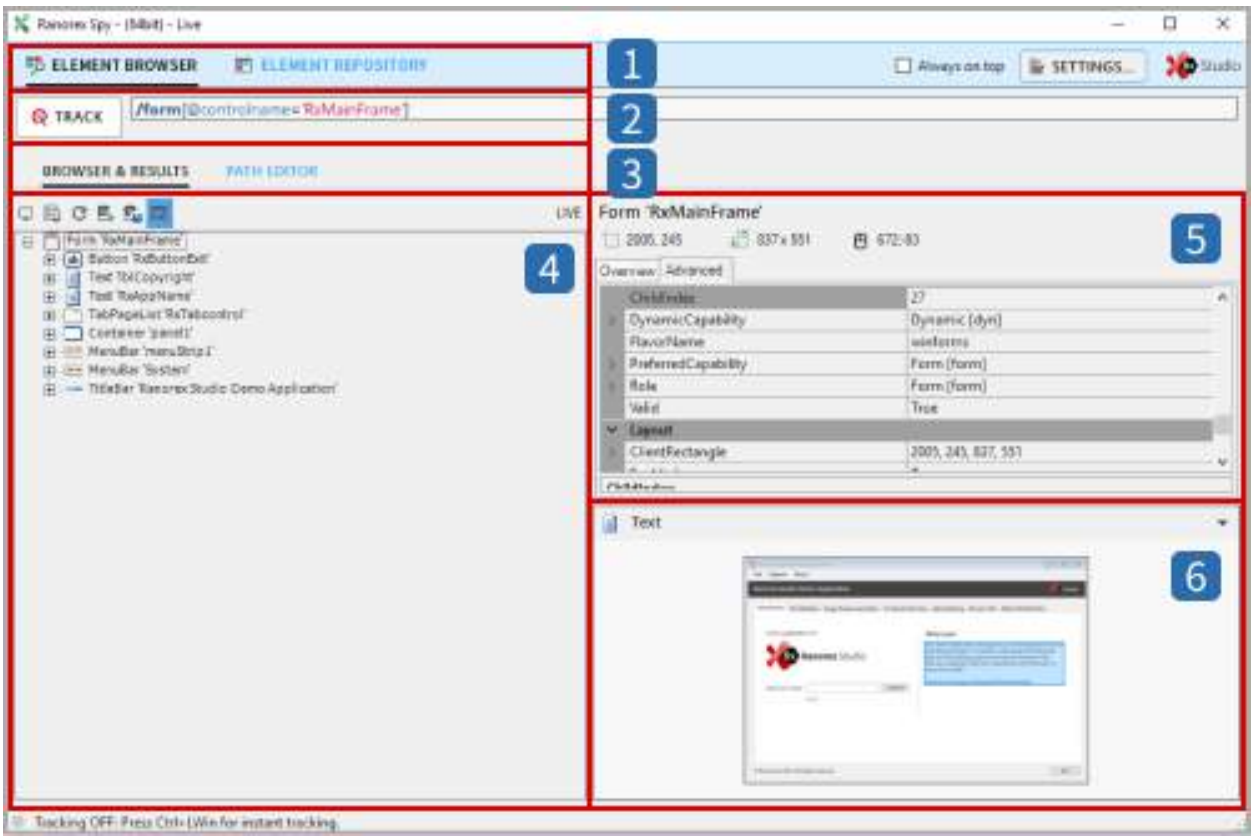
Ranorex Spy is available as a 32-bit and as a 64-bit version. When you run Spy from within Ranorex Studio, it uses the same bit architecture as Ranorex Studio. When you run the standalone version, it's up to you which bit architecture you choose.

We recommend always using all Ranorex Studio tools with the same bit architecture as your AUT.

Ranorex Spy layout

In the image below, you can see the layout of the standalone Spy with the BROWSER & RESULTS tab active. The areas framed in green change when selecting PATH EDITOR instead. These areas are explained separately as part of the pages ⇢ Browser & results and ⇢ Path editor.

The areas in red are fixed and explained further below.



1 **Element browser / repository selector (standalone only)**
Switches between the element browser and the repository. Loading repository files (.rxrep) is only possible in the standalone Spy.

2 **Track button** and **RanoreXPath field**
The track button lets you track, identify, and add new UI elements. The RanoreXPath field displays the RanoreXpath of an element and also lets you edit it.

3 Switch between **browser & results screen** and the **path editor**
These are explained separately on the pages⇢ Browser & results and ⇢ Path editor.

4 **UI-element tree view**
This hierarchical tree structure represents all running applications (depending on whitelist settings) and their UI structure.

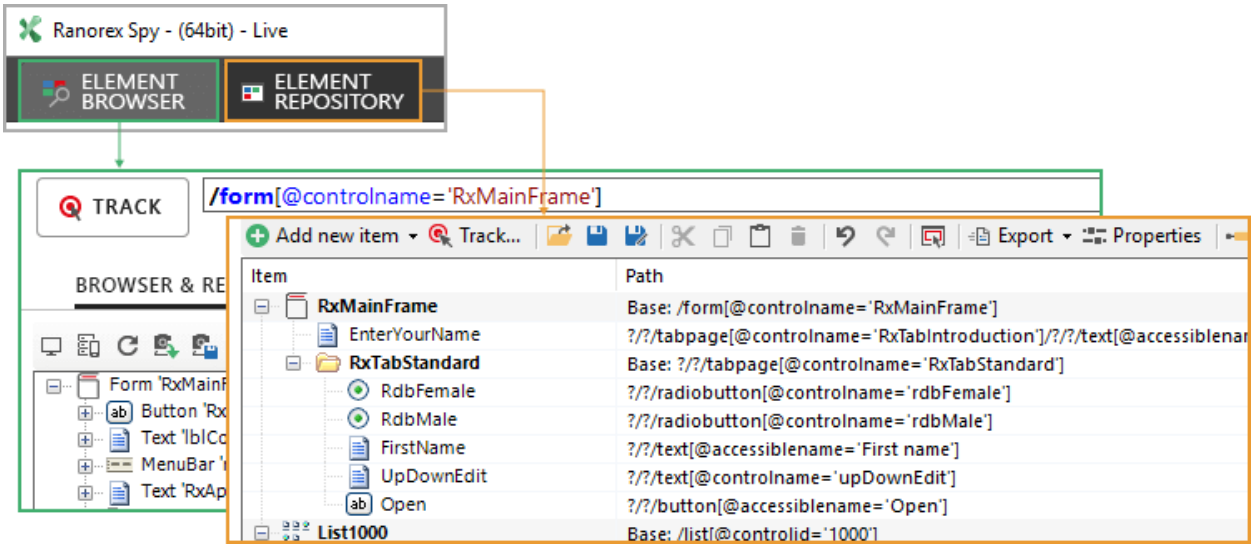
5 **UI-element details**
Selecting a UI element in the tree displays detailed information about it in this section.

6 **Image navigation area**
Displays a selected UI element as it would appear in the application and can be used to navigate through the UI.

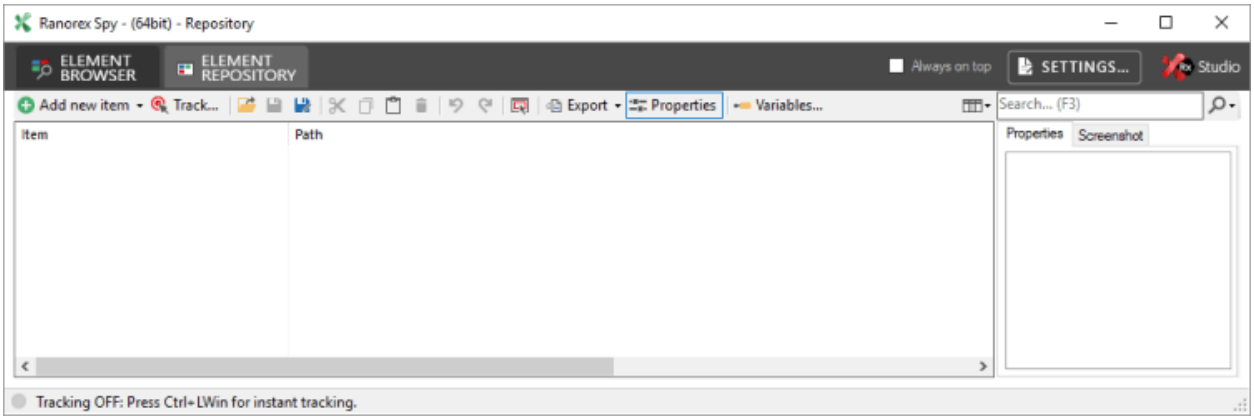
Element browser/repository selector

Only available in the standalone version of Spy. Lets you switch between the element browser and the repository view.

In the repository view, you can load and manage repository files (.rxrep). This is particularly useful when you want to add repository items to a repository on a computer where Ranorex Studio isn't available.



The repository view is the same as in Ranorex Studio and has all the same functions.



> **Reference**
>
> Managing repositories is explained in Ranorex Studio fundamentals > ⇢ Repository.

Track function

Ranorex Spy has a track function that lets you identify UI elements and add them to the element tree, from where you can then add them to your repository.

1. **TRACK** button and **RanoreXPath** field:
   - The track button starts the tracking process.
   - The RanoreXPath field displays the RanoreXPath of the tracked UI element

2. **Root UI element**
   - You can only track UI elements that are contained in the root element.
   - In the example, the RxMainFrame (The Ranorex Studio Demo Application) is the root element and the button `RxButtonExit` is the tracked UI element. This means that tracking a button in the Windows Calculator would not work because it's a separate application. If the root was the entire endpoint, it would work.

RanoreXPath field

1 ***RanoreXPath*** *field in* ***edit*** *mode*

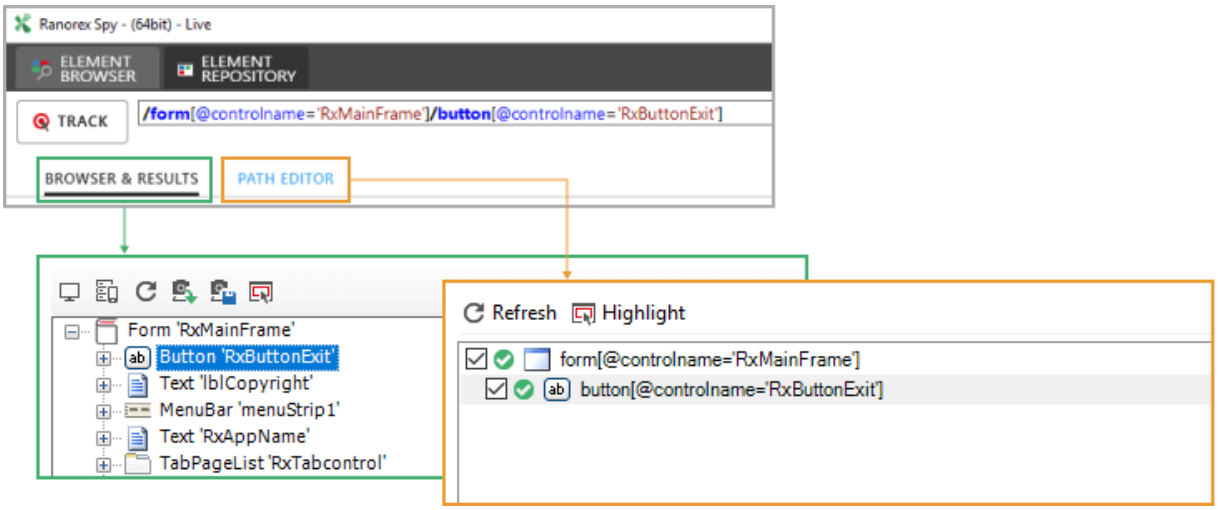
- Normally, the RanoreXPath field is white and displays the RanoreXPath of the element currently selected in the tree.
- When you click in the field and press Enter, it changes to edit mode. This is indicated by a red X, yellow background, and a message below the field.

2 **Exit edit mode**

- Clicking the red **X** **exits edit mode.**

Working environment selector

The working environment selector lets you switch between the browser & results screen and the path editor.



Browser & results screen

As explained on the introductory page to this chapter, Ranorex Spy has two working environments: the browser & results screen and the path editor. On this page, you'll find out how the former works and what it is used for.

Screencast

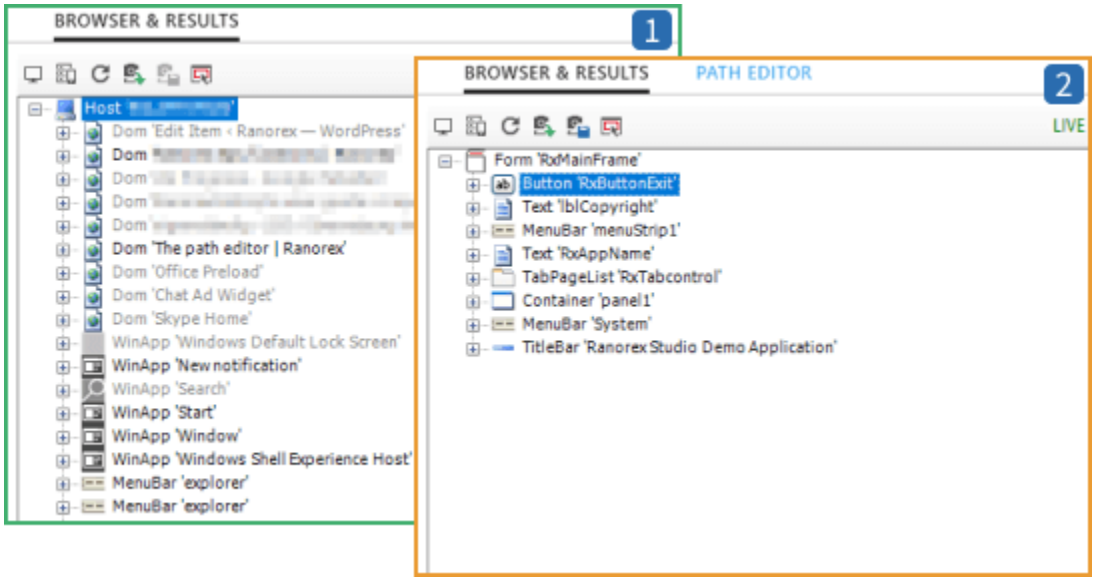
The screencast “Spy functions” walks you through the information found in this chapter:

Watch the screencast now

Element tree browser

The element tree browser is on the left side of the working environment by default. It’s a hierarchical representation of running applications and their UI elements. By default, it displays all applications that are running on an endpoint (i.e. machine) and whitelisted in Ranorex Studio. However, you can also display only a certain application or node.

The functions of the tree browser are explained further below.



1 Element tree browser displaying **ALL** running and whitelisted applications.

2 Element tree browser displaying only the application `RxMainFrame`, which is the Ranorex Studio Demo Application.

Element tree browser functions

The area above the element tree browser contains several buttons and a status indicator. The button layout differs slightly between the standalone and the integrated Spy. The difference is indicated below in the captions.

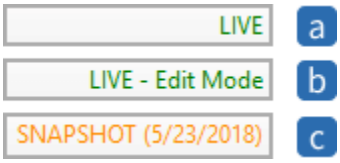
1 **Browse endpoint** and **Endpoints** (only in standalone Spy)

2 **Refresh**

3 **Load from snapshot…** and **Save as snapshot…**

4 **Highlight elements**

5 **Status indicator**
Displays the status of the element tree browser. Three statuses are possible:



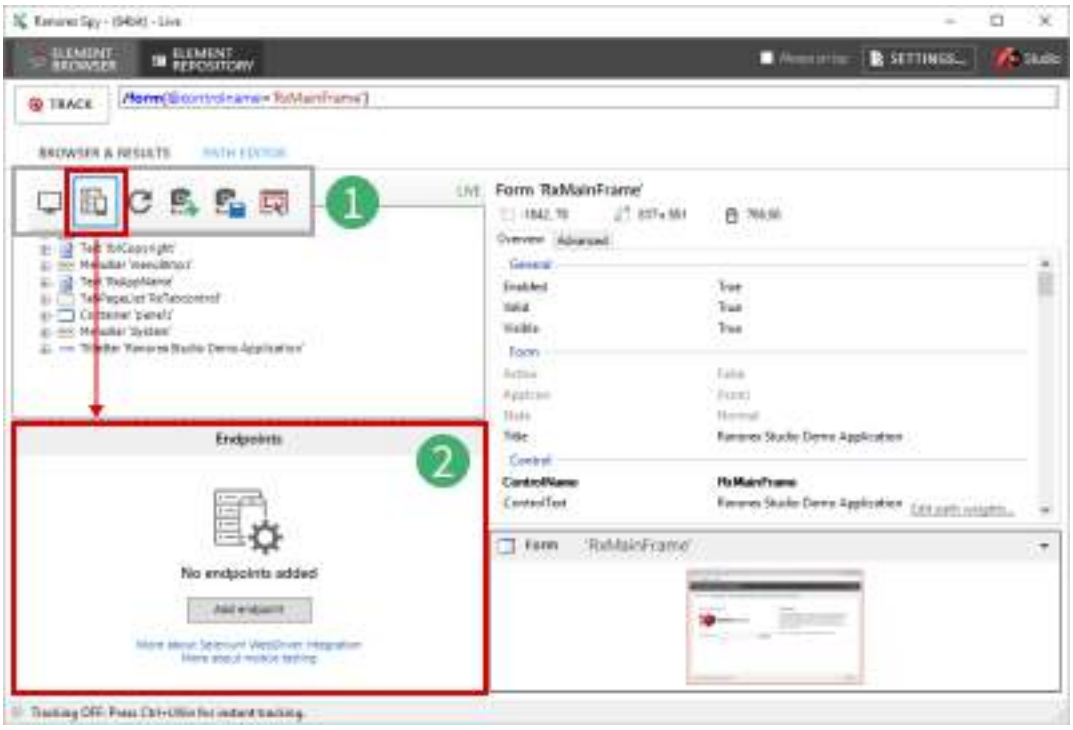
a **Live display mode**
The tree represents the applications of the currently active endpoint.

b **Live edit mode**
You are editing an element of the currently active endpoint.

c **Snapshot mode**
The tree was loaded from a Ranorex snapshot, which represents a static snapshot of an endpoint's applications.

Browse endpoint and Endpoints

- **Browse endpoint** displays **all** currently running and whitelisted applications on the active endpoint (= automation root).
- If you've restricted the tree to a specific application, click **Browse endpoint** to display all applications again.
- If no endpoints are defined, the automation root is the current host, i.e. the machine Spy is opened on.
- **Endpoints** (only available in standalone Spy) opens the endpoints pad, from where you can select which endpoint you want to use as automation root. For the integrated Spy, you need to set the automation root in the endpoints pad in Ranorex Studio.



1. **Click** the **Endpoints** button in the toolbar.
2. The endpoints pad opens.

> **Further reading**
>
> Endpoints are explained in Web and mobile testing > ⇢ Endpoints.

Refresh

- Updates the element tree to reflect changes in the applications.

Load from snapshot… and Save as snapshot…

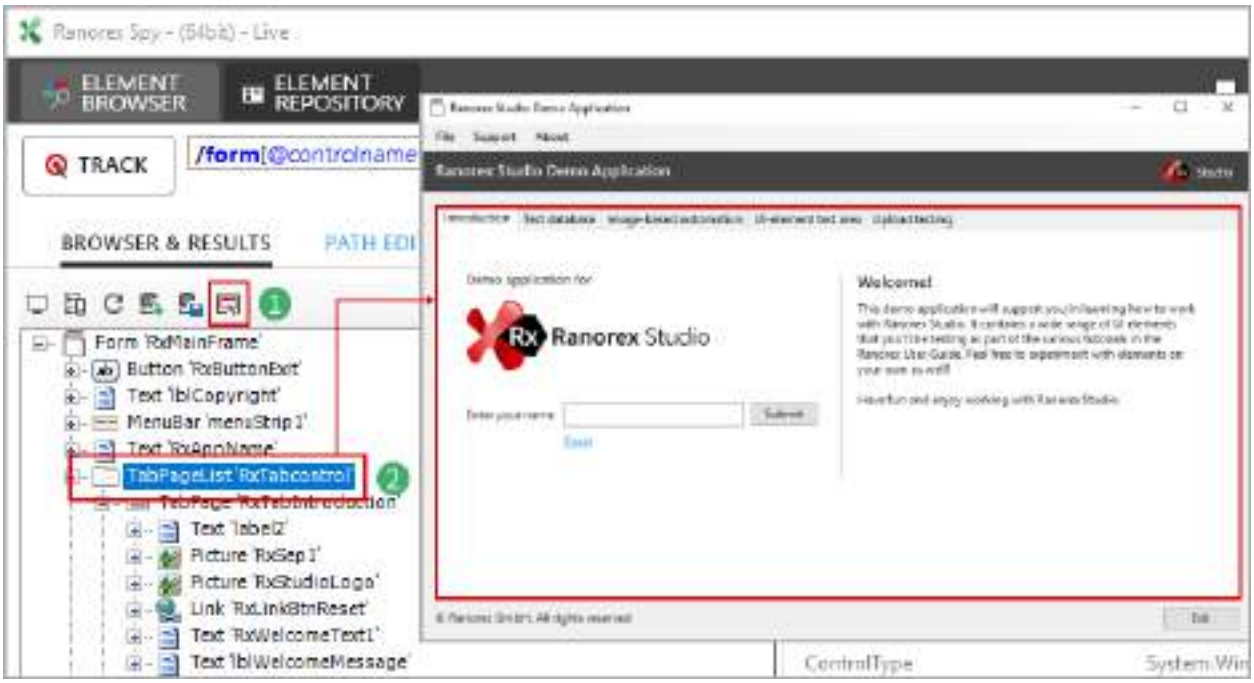
- **Load from snapshot** lets you open an existing **snapshot file.**
- **Save as snapshot** saves the currently selected UI element and all of its ancestors and descendants to a **snapshot file.**

Further reading

Ranorex snapshots are explained in Ranorex Studio advanced > Ranorex Spy > ⇢ Snapshot files.

Highlight elements

This function is the same as the one in the repository toolbar. It highlights the selected element in the UI of the actual application with a red frame. Naturally, the application must be running for this to work.



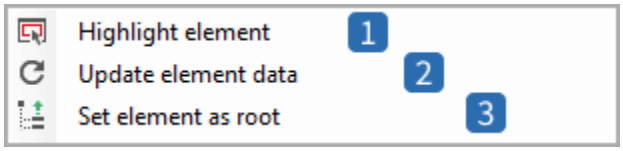
1. **Click** the **Highlight elements** button.
2. The element is highlighted with a red frame in the application.

Note

The respective application must be running and visible on the endpoint for this to work.

Element tree browser context menu

In addition to the toolbar buttons, right-clicking an element in the tree brings up a context menu that offers many further options.

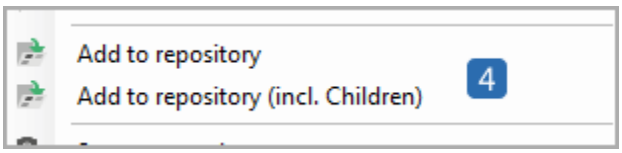


1 **Highlight element**
Highlights the element in the application. Explained in the previous topic.

2 **Update element data**
Refreshes the element. Explained in the previous topic.

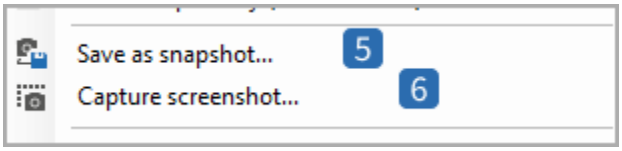
3 **Set element as root**
Makes the element the root of the element tree browser.

1. In the tree, **select** the **UI element** you want to set as root.
2. In the context menu, **click Set element as root.**
3. The tree now has the element as root.



4 **Add to repository**

- Generates a repository item for the element and adds it to the repository.
- You can either add only the selected element, or the element and all its children.
- For the integrated Spy, the repository item is added to the repository currently active in Ranorex Studio.
- For the standalone Spy, the repository item is added to the default Spy repository or one that you've loaded.



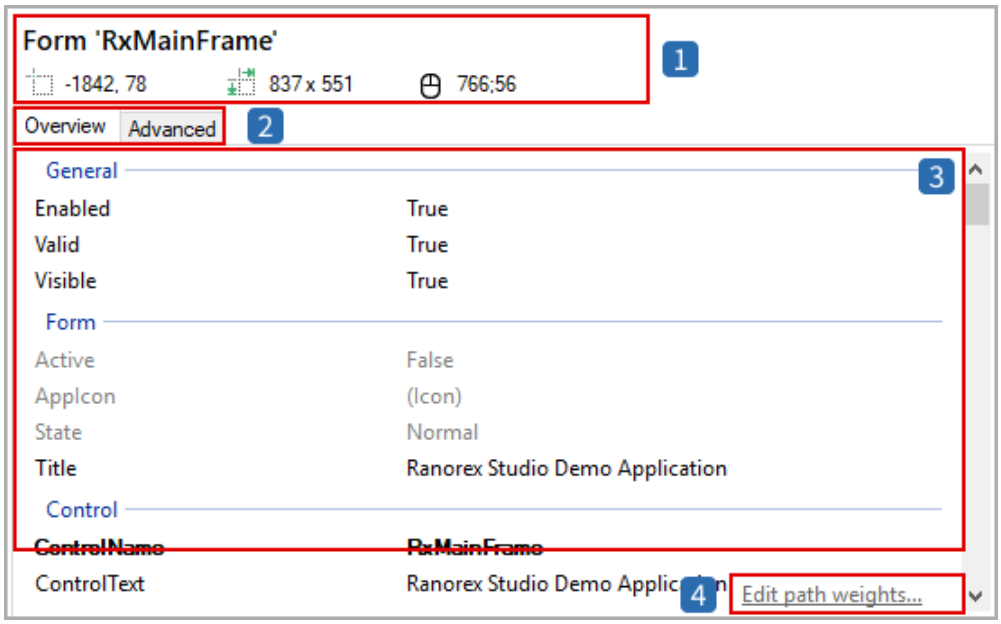
5 **Save as snapshot…**
Saves the element to a snapshot file. Explained in the previous topic.

6 **Capture screenshot**

- Captures a screenshot of the element as it appears in the application. Useful for image-based automation.
- The application containing the element must be running and the path to the element valid. Refresh the element tree beforehand if necessary.
- After it has been captured, the screenshot is opened in the image editor.
- Saved screenshots can also be used for image comparison in code, e.g. for finding and comparing images or image-based automation.

UI element details

The details area lists all properties and attributes of a selected UI element.



1 ***Primary adapter*** *and* ***name*** *of element*

- Additionally, information about the size and position of the UI element on the desktop are displayed.
- Values are in pixels and based on an x/y coordinate system.

2 Switch between the overview and advanced details.

3 Area where the details are displayed.

> **Further reading**
>
> UI elements and their details, i.e. properties and attributes, as well as the concept of adapters are explained in Ranorex Studio advanced > ⇢ UI elements.

4 **Edit path weights...**
Opens a menu that allows you to configure the mapping of dynamic UI elements.

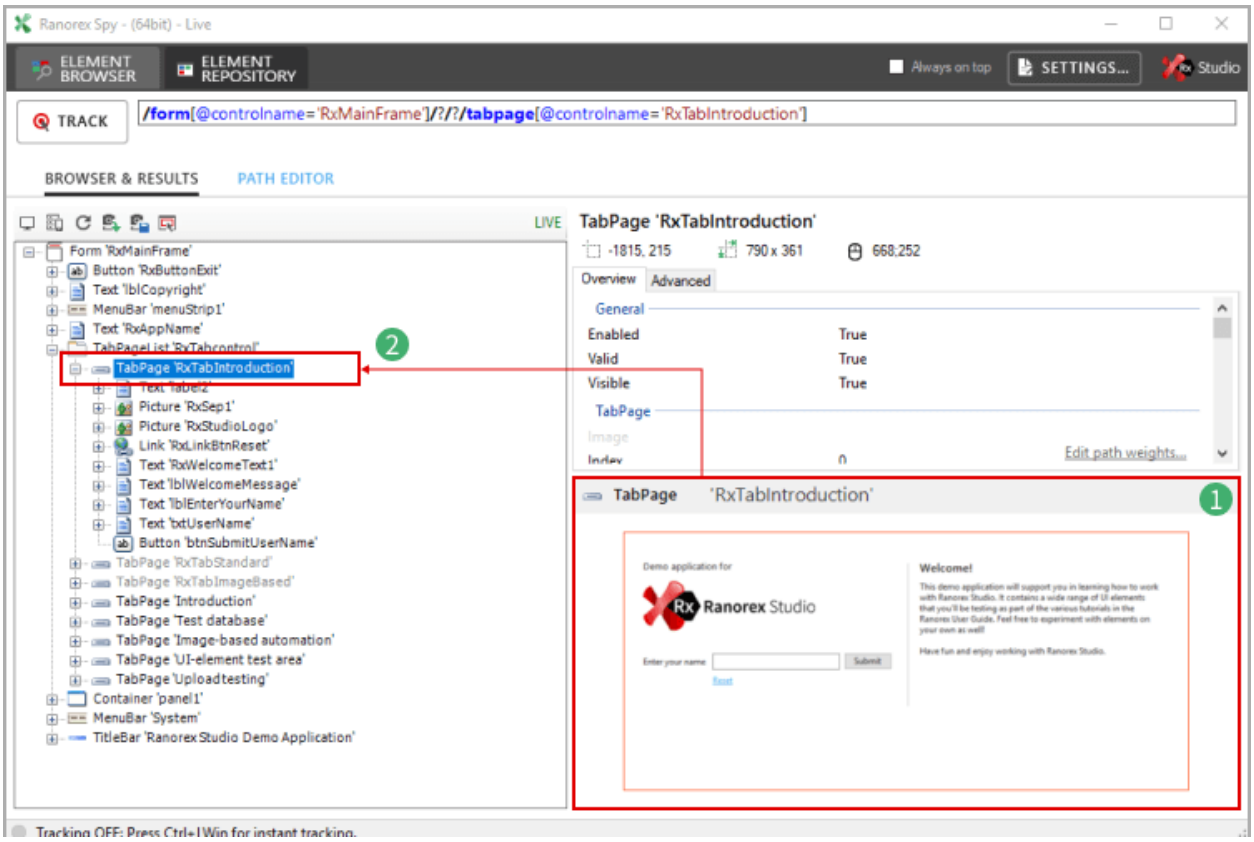
> **Further reading**
>
> Mapping dynamic UI elements is explained in Ranorex Studio expert > ⇢ Mapping dynamic UI-elements.

Image navigation area

The image navigation area shows the UI the selected element is part of. You can navigate through the UI as if you were in the application. When you click a UI element in the image, it will be selected in the tree.

1 **Click** a UI element in the image.

2 The corresponding tree element is selected.



- At the top of the image navigator, you can see the primary adapter type and the name of the currently selected UI element.
- When you move the mouse over a UI element in the image, the adapter type and name will change accordingly.
- Double-clicking outside the UI image switches to the image for the parent element.

The path editor

The path editor is the second main working environment in Ranorex Spy. It's where you specify and edit the RanoreXPath of identified UI elements from the element tree. The path editor has several helpful functions to assist you in creating RanoreXPaths that fit your needs.

On this page, you'll find out how the path editor works and what functions it offers.

Screencast

The screencast “The path editor” walks you through information found in this chapter:

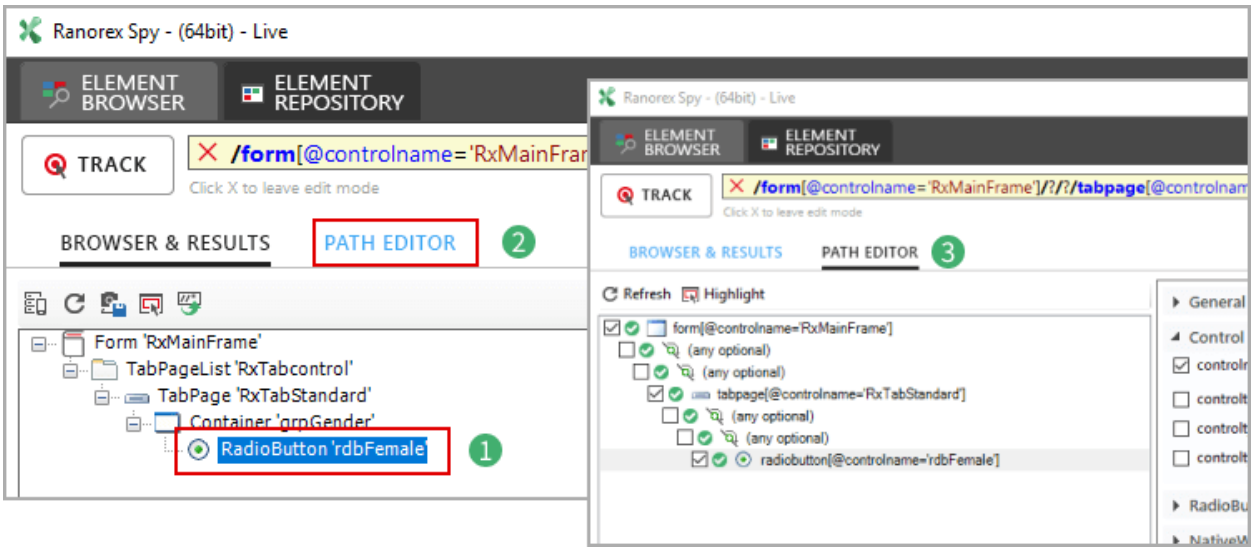
Watch the screencast now

Access the path editor

You can access the path editor in two ways: from the working environment selector within Spy, or directly from an opened repository.

From Spy

1. In the element tree, **select** a UI element.
2. In the working environment selector, **click PATH EDITOR**.
3. The path editor working environment appears with the UI element highlighted in the path tree section.



From a repository

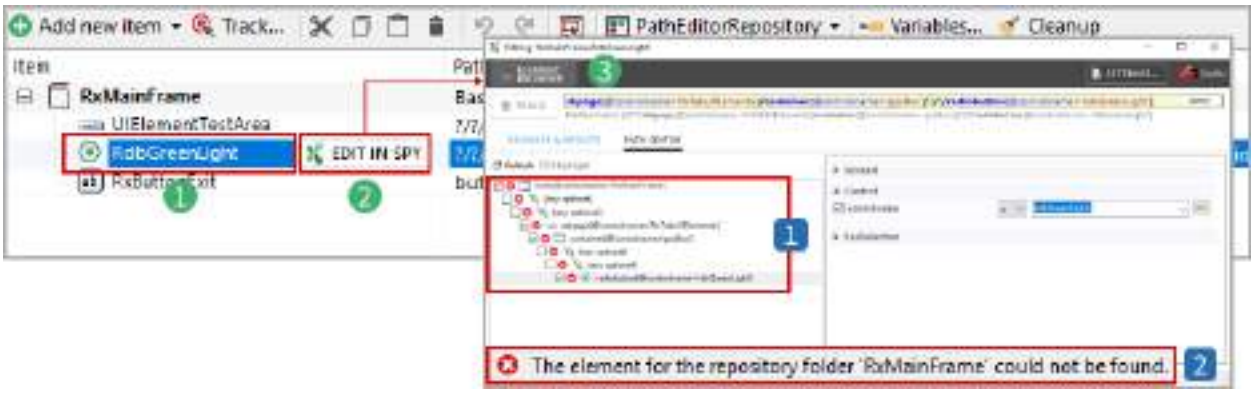
1. In a repository, **select** a repository item and…
   a. …**click EDIT IN SPY.**
   b. **…right-click** it and **click Edit in Spy…**.

Element could not be found

A common error message that occurs when opening a UI element in the path editor as explained above is that the element couldn't be found. This usually happens because the application containing the UI element isn't currently running.

Initial situation

1. In a repository, **select** a repository item.
2. **Click EDIT IN SPY**.
3. The path editor opens with an error message and the path tree displays red marks, indicating that the respective UI elements specified by the RanoreXPath could not be found.



1. Red symbols indicating that the respective UI elements specified by the RanoreXPath could not be found.
2. Error message in the status bar of Spy indicating that the UI element could not be found.

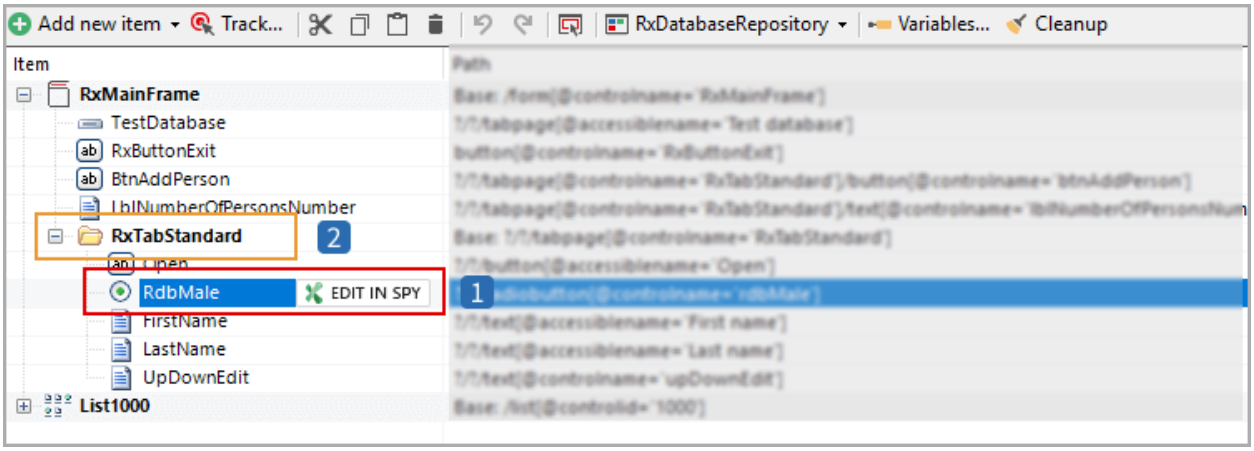
Solution

1. **Start** the application containing the UI element(s).
2. In the path editor, **click** the **Refresh** button to update the references to the UI elements.

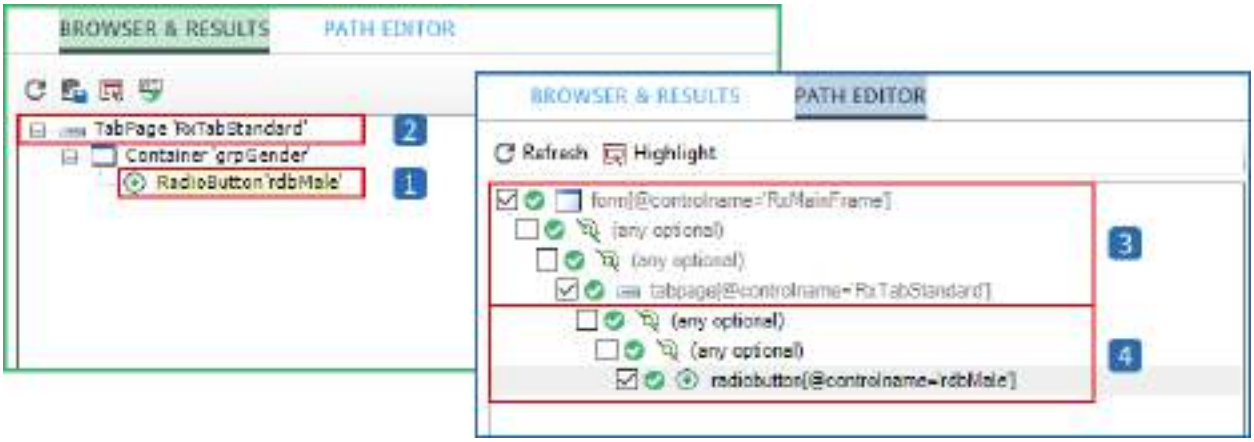
Locked tree steps

When opening a repository item in the path editor, its parent repository item limits what part of the RanoreXPath you can edit and what UI elements you can track. Any components of the RanoreXPath that specify the parent and above **cannot be edited**. UI elements at the same level or above the parent **cannot be tracked**.

This is because changes at these locked levels would affect other repository items and potentially break existing RanoreXPaths. If you need to make changes to the parent repository item or above, edit those items specifically.



1. **EDIT IN SPY** opens the repository item for editing in the path editor.
2. The changes you can make to the RanoreXPath are limited by the **parent repository item**.



1 The browser & results working environment displays the repository item in the element tree.

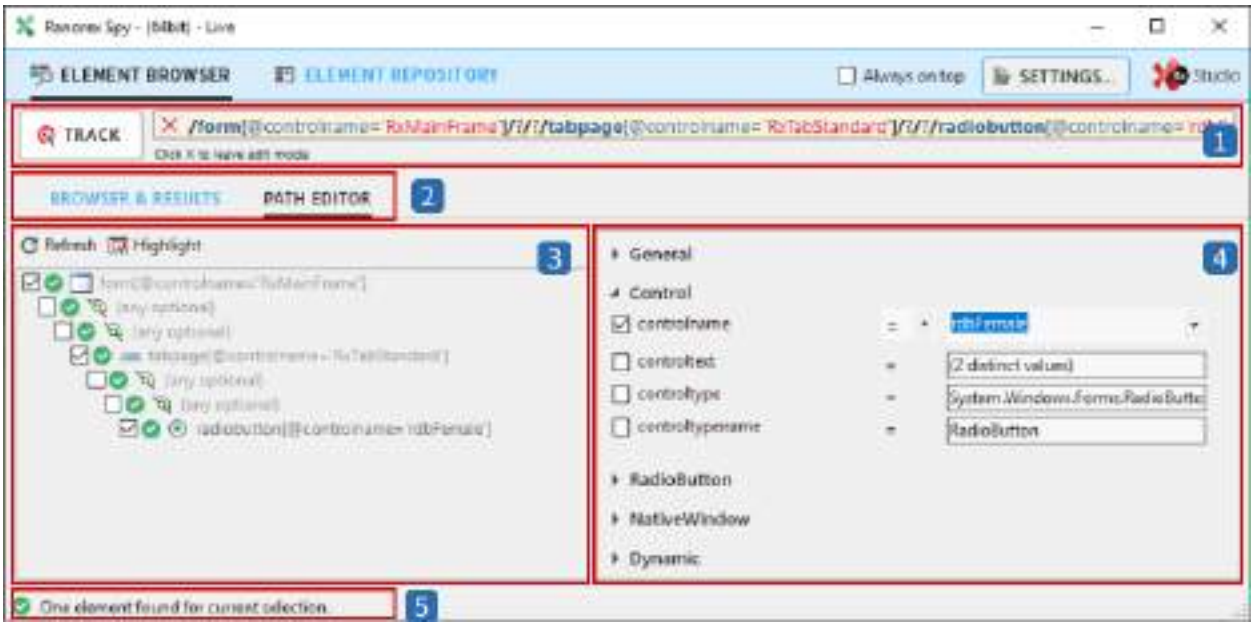
2 The parent of the repository item has been set as the **root element** automatically**.** This element represents the limit for tracking and editing.

3 In the path editor, all elements **above and including** the root element are **locked** for tracking and editing.

4 All items **below** the root element **can** be tracked and edited.

Path editor working environment

The path editor working environment consists of the two basic Spy sections (red frame), two specific path editor sections (green frame), and a status indicator (orange frame).



1 **Track button** and **RanoreXPath** field. Explained in >Introduction.

2 **Working environment selector**

3 **Path tree**
Displays the components of a UI element's RanoreXPath in a tree structure.
Refresh: Updates the tree to reflect changes in the respective application.
Highlight: Highlights the UI element in the respective application.
Both fail if the application isn't running.

4 **Attribute specification**
Lists available UI-element attributes, operators, and values for use in the RanoreXPath.

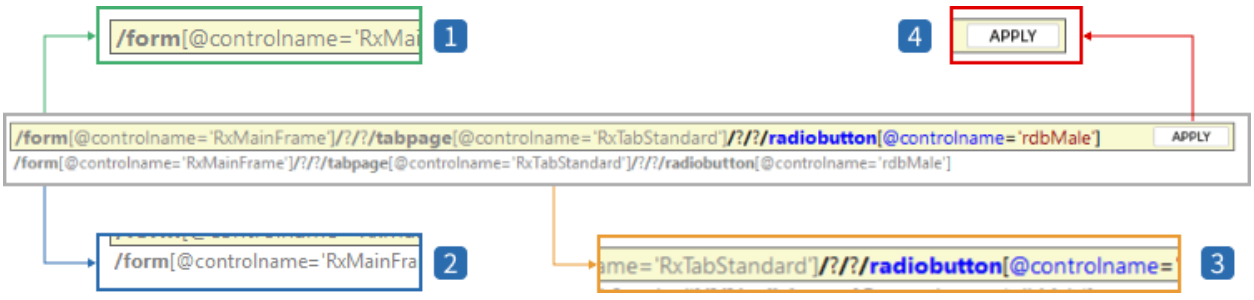
5 **Status indicator**
Indicates how many UI elements the current RanoreXPath identifies.

Edit the RanoreXPath

The main application of the path editor is to support you in editing the RanoreXPath so as to make it more general or specific, depending on your needs. The section where you make the actual changes to the path is the RanoreXPath field. We've touched on it in the introduction to this chapter, but we'll go into more detail here. **Depending on whether you edit a UI element from the element tree or an already existing repository item from a repository, editing is a little different**.

When editing UI elements from the element tree, the path editor functions more like a search and allows you to easily see what UI elements a given RanoreXPath will identify. You can consider it a sandbox environment that doesn't affect your test, unless you start adding identified UI elements to your repository.

When editing existing repository items, you change your test. Therefore, the RanoreXPath field is a little more complex when doing so, as shown in the image below:



1 The main yellow **field** is where you edit the RanoreXPath.

2 The line below it displays the currently applied (valid) RanoreXPath specification, i.e. as it was before you started making changes.

3 Parts of the RanoreXPath may be grayed out in the yellow field. This indicates they are locked and cannot be edited (see topic Locked tree steps further above).

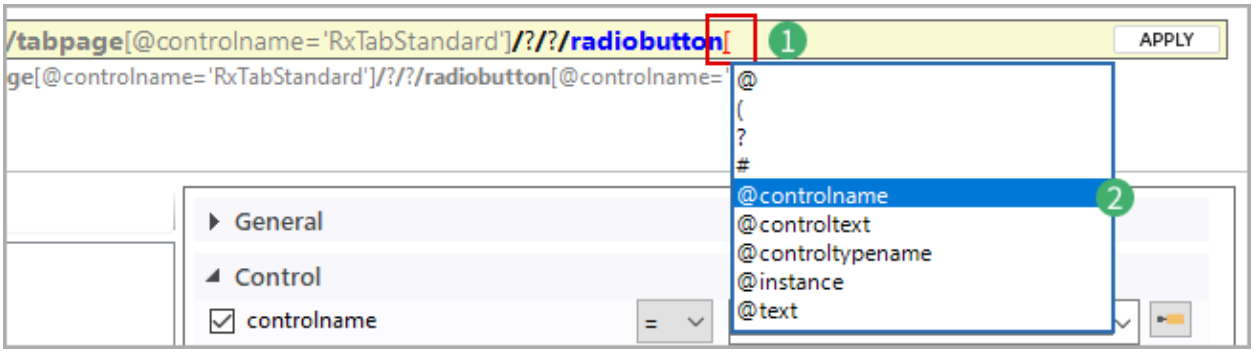
4 The **APPLY** button replaces the currently applied RanoreXPath (lower line) with the currently edited RanoreXPath (yellow field), closes Spy and returns to Ranorex Studio, where you will see the new path reflected in the repository item.

RanoreXPath syntax support

RanoreXPath syntax is fairly complex. The path editor has built-in syntax support to make editing easier.

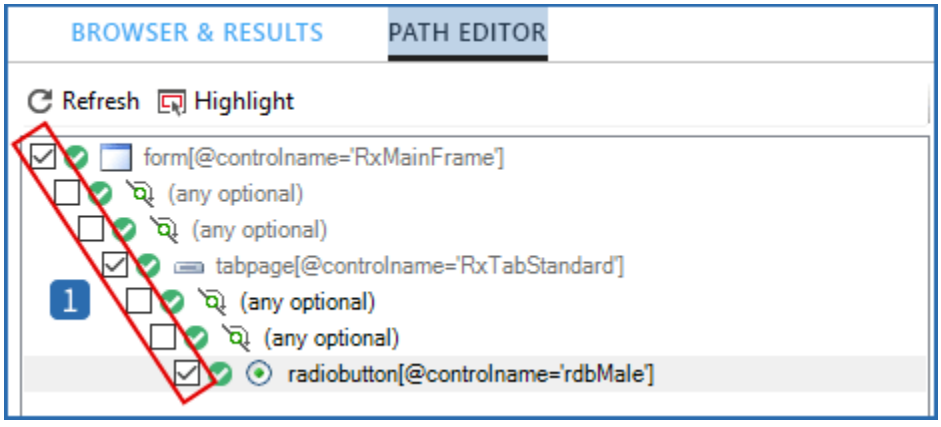
1 While editing aRanoreXPath, **press** Ctrl + Space .

2 A drop-down list opens with semantically possible RanoreXPath syntax parts.



Selecting/deselecting path components

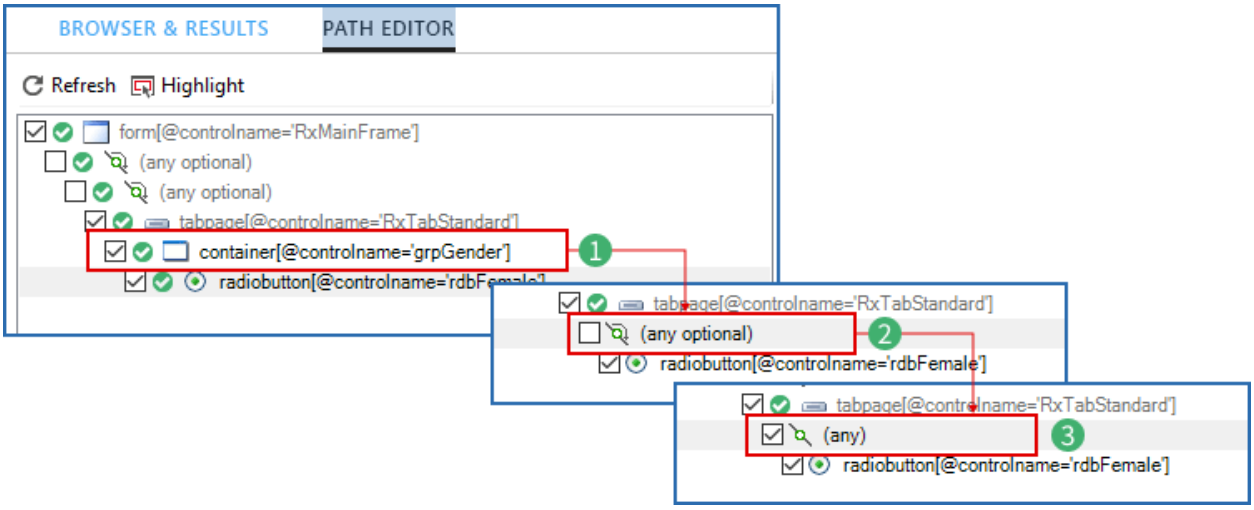
Using the path tree, you can easily include or exclude specific components from a RanoreXPath. Simply select or deselect them and the path will be updated automatically. However, there are some limitations (see below).



1 **Checkboxes** to select/deselect path components

Try the following to see the effect this has:

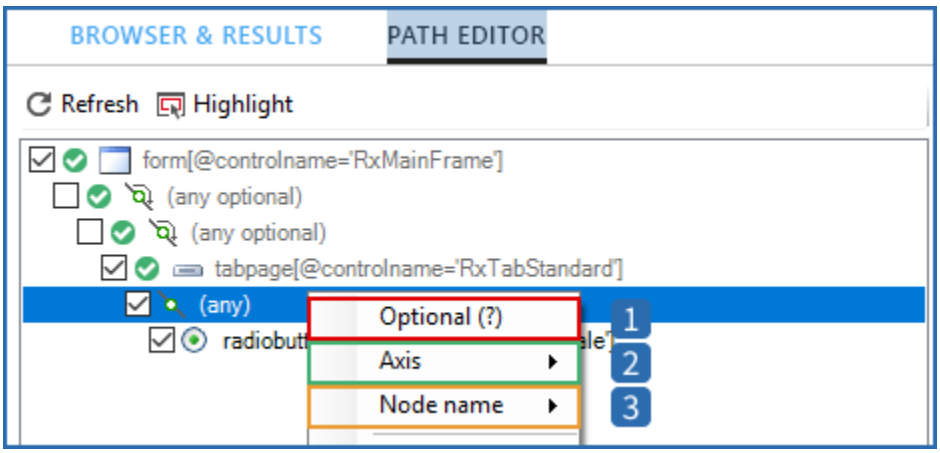
1. **Select** a component. It's included in the RanoreXPath.
2. **Deselect** the component. It's excluded from the RanoreXPath, but also loses information, i.e. the adapter type and attribute.
3. **Re-select** the component. The lost information is not recovered; the component is now a placeholder for any UI elements between the tabpage and radiobutton .



If information is lost this way, you can either retrack the UI element or add the information again manually.

Tree context menu

The context menu for components in the path tree lets you specify them in several ways:



1. **Optional (?)**
   Declares the component as optional, indicated by a ? in the RanoreXPath before the modified component. When a component is optional, it isn't required for the path to work, i.e. the path will still work if nothing is found for the component. When placed between an adapter and an attribute, both become optional, meaning the path will work if both, just one, or neither identify a UI element.

2 **Axis**
Axes are relationship operators. They allow you to select UI elements by way of their relation to other UI elements.

Possible axes:

child	Selects all children of the current node.
descendant-or-self	Selects all descendants (children, grandchildren, etc.) of the current node, and the node itself.
ancestor	Selects all ancestors (parents, grandparents, etc.) of the current node.
self	Selects the current node.
descendant	Selects all descendants (children, grandchildren, etc.) of the current node.
parent	Selects the parent of the current node.
ancestor-or-self	Selects all ancestors (parents, grandparents, etc.) of the current node, and the current node itself.
preceding-sibling	Selects all siblings before the current node.
following-sibling	Selects all siblings after the current node.

Further reading

Optional components and axes are part of RanoreXPath and explained in Ranorex Studio advanced > ⇢ RanoreXPath.

3 **Node name**
Allows you to assign a specific adapter to an element.

Further reading

Adapters, attributes, and values are explained in Ranorex Studio advanced > ⇢ UI elements

Snapshot files

A **Ranorex snapshot** is a file representation of the user interface (UI) structure of an AUT at a particular point in time. A Ranorex snapshot captures all interface elements, their hierarchy, values, etc.

Snapshot files are created viewed in Ranorex Spy. Their file extension is `.rxsnp`.

Typically, snapshot files are used to share detailed information about the UI of an application with the Ranorex support or technical sales team. They are also helpful when asking for technical help on our forums.

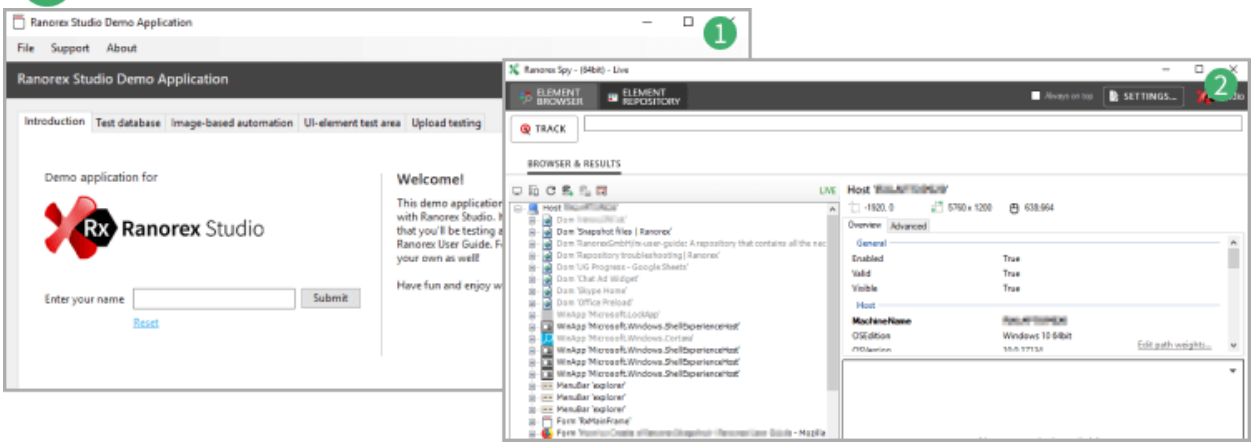
> **Screencast**
> The screencast "Snapshot files" walks you through information found in this chapter.:
>
> Watch the screencast now

Create a standard snapshot file

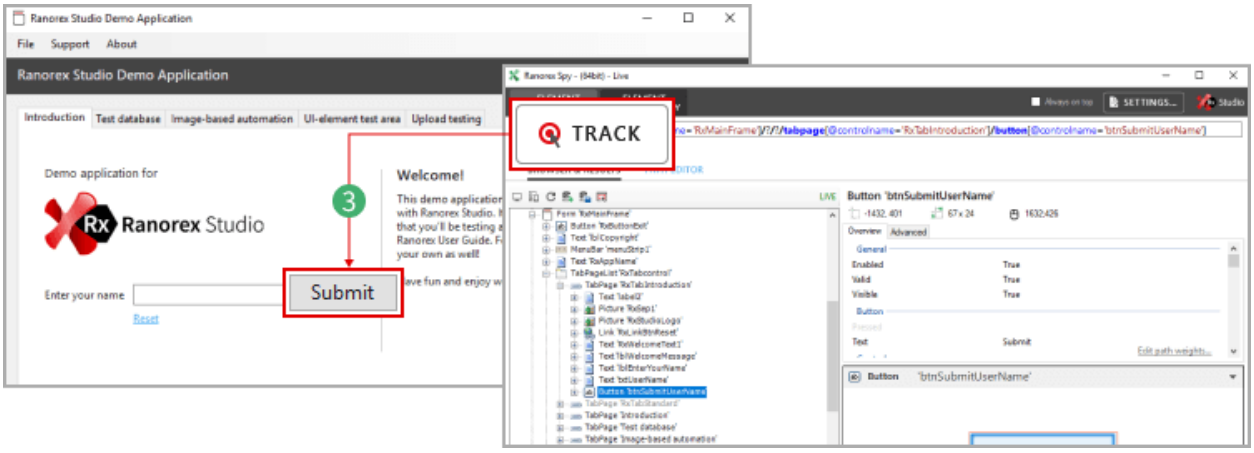
Standard snapshot files capture a UI that contains no hidden UI elements like menus, menu items, context menus, tool tips, etc. Creating a snapshot of a UI that contains these elements is explained in the next topic below.

To create a standard snapshot file:

1. **Start** your AUT (e.g. the Ranorex Studio Demo Application).
2. **Start** Ranorex Spy.

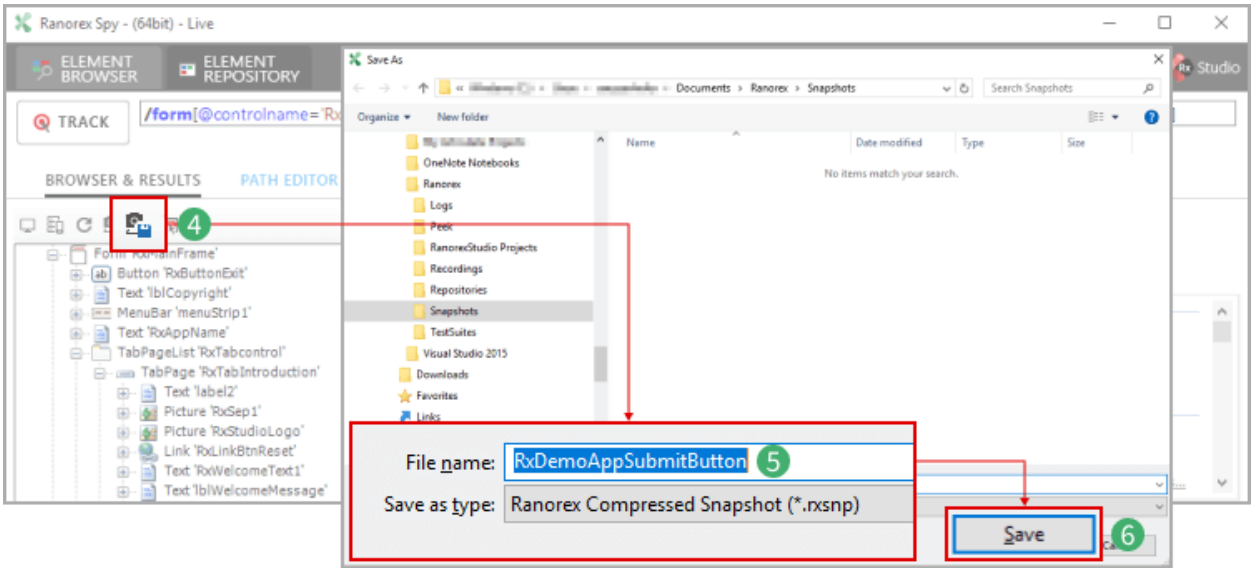


3. Using the **TRACK** button**, track** a UI element in the AUT (e.g. the **Submit** button as shown below).

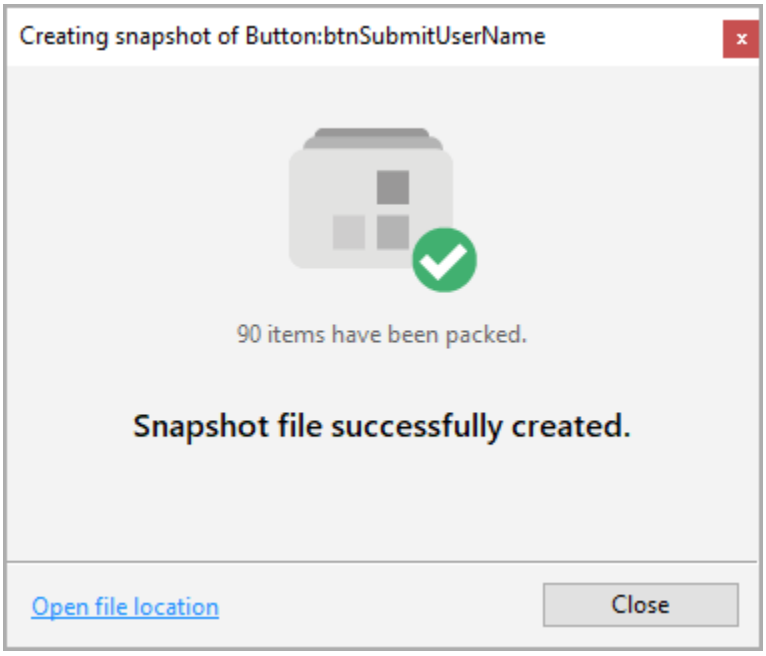


4. In the toolbar of the element tree browser, **click** the **Save as snapshot…** button.
5. **Name** the file and **specify** where it should be saved.
6. **Click Save** to start snapshot file creation.

> **Note**
>
> The default save location for snapshot files is `USERPROFILE%\RanorexSnapshots`.



7 While the snapshot is being created, a progress bar indicates progress. Upon completion, the following message appears. **Click Close.**



Note

Snapshots **usually contain** the **tracked/selected UI element** and **the complete ancestor subtree, i.e. all elements of the respective application**. You can change this default behavior under Settings > Advanced > Let snapshot contain complete ancestor subtree.

Further reading

Advanced settings are explained in Ranorex Studio system details > Settings & configurations > ⇢ Advanced settings & configurations.

Create a snapshot of hidden UI elements

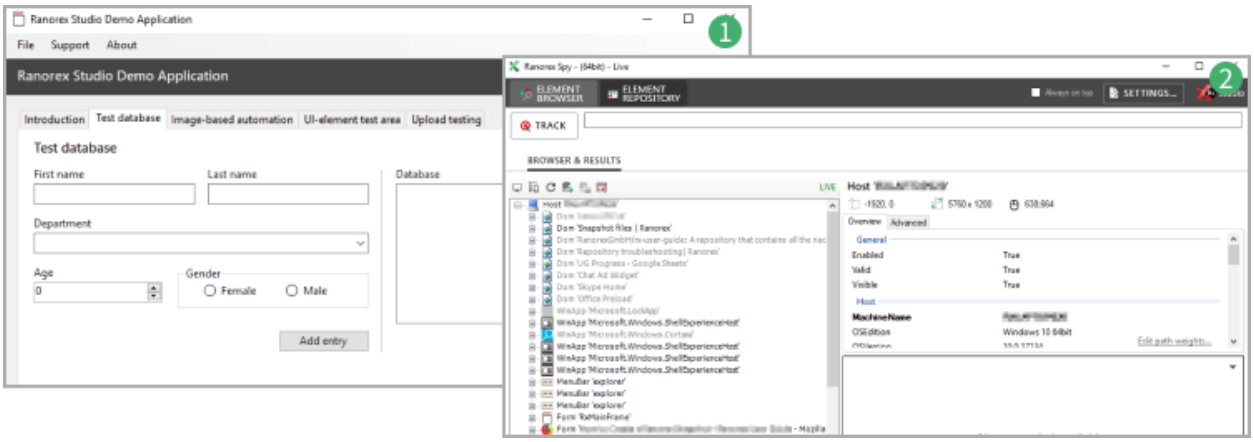
Some UI elements, such as **drop-down menus**, **pop-up windows**, **combo boxes** etc., only become visible after an interaction like a click and usually disappear if the AUT loses focus. These elements are “hidden” and therefore **not automatically included** in a snapshot file. This is because they appear as separate items in Ranorex Spy at the top level of the element

tree. Capturing these elements in a snapshot file requires use of the **instant tracking function**.

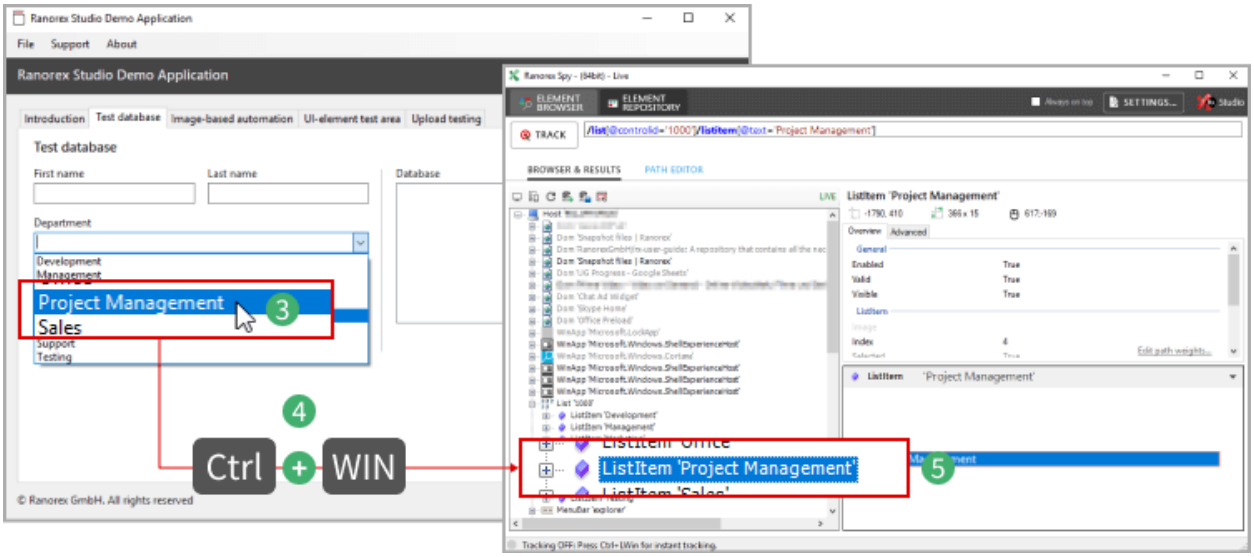
> **Further reading**
>
> Tracking hidden UI elements is explained in Ranorex Studio advanced > Tracking UI elements > ⇢ Instant tracking.
> An alternative method is explained in Ranorex Studio advanced > Tracking UI elements > ⇢ Track button.

To create a snapshot file with hidden UI elements:

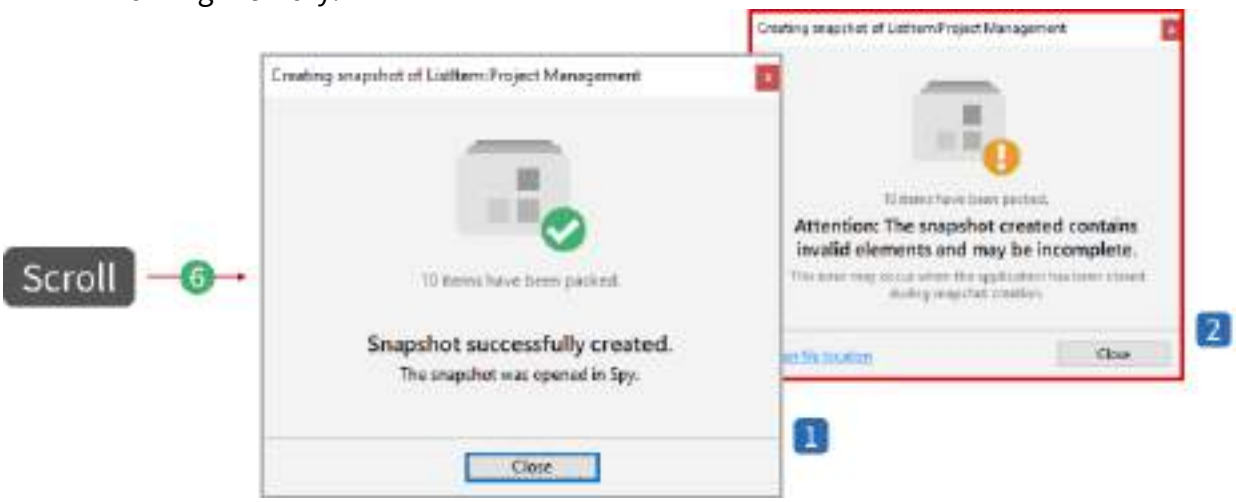
1. **Start** your AUT and **navigate** to where the hidden UI element you want to track appears (e.g. in the Test database tab of the Demo Application).
2. **Start** Ranorex Spy.



3. **Make** the hidden UI element **visible**, e.g. by opening the containing drop-down menu, as for the Project Management list item in the Demo Application, and **ensure** it is in focus, e.g. by mousing over it.
4. With the UI element in focus, **press** Ctrl + WIN.
5. The tracked UI element appears in the element tree in Spy.



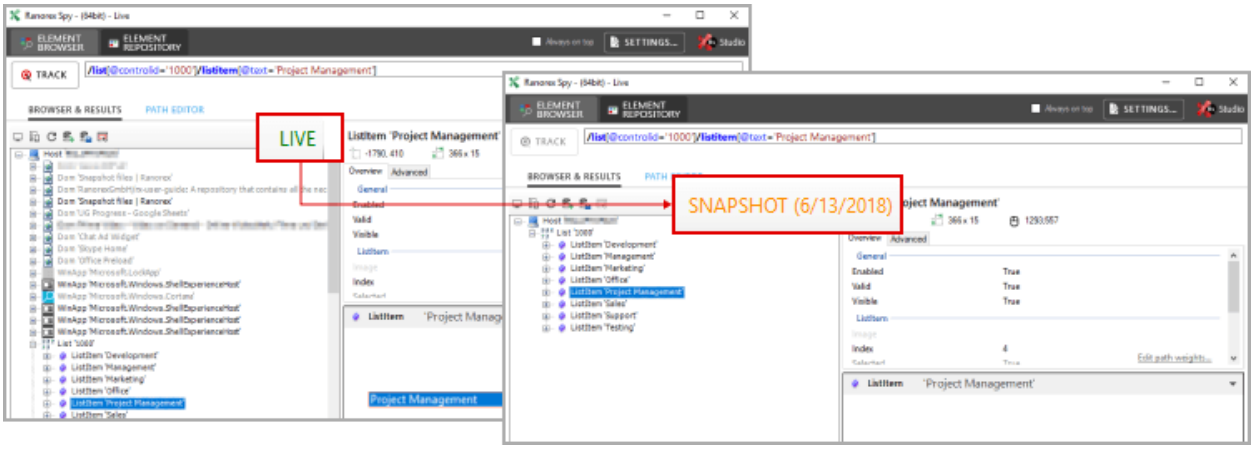
6 Now immediately before anything else, **press** Scroll to create a snapshot of the previously tracked hidden UI element and its ancestor subtree and cache it to the working memory.



1 Snapshot file created successfully and cached to working memory. In this case, 10 UI elements were packed into the snapshot file, which is now opened in Ranorex Spy.

2 **Warning** that the snapshot file may be incomplete. This usually happens when you try to create a snapshot of hidden UI elements without using Scroll.

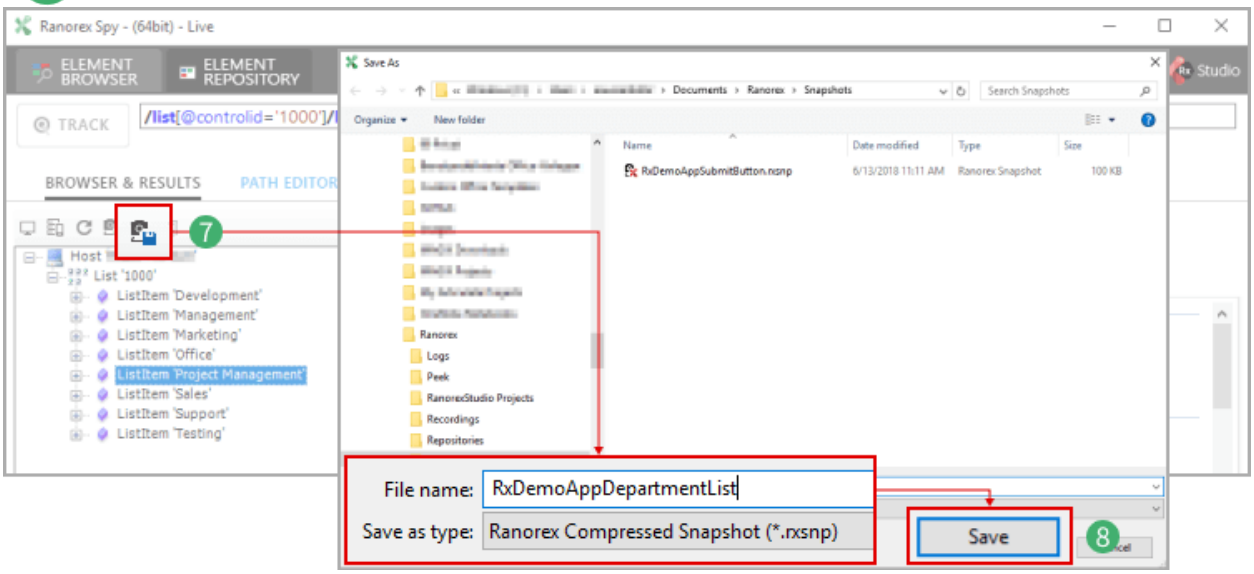
7 The snapshot file is automatically opened in Ranorex Spy. This is indicated by the status message that switches from LIVE to SNAPSHOT.



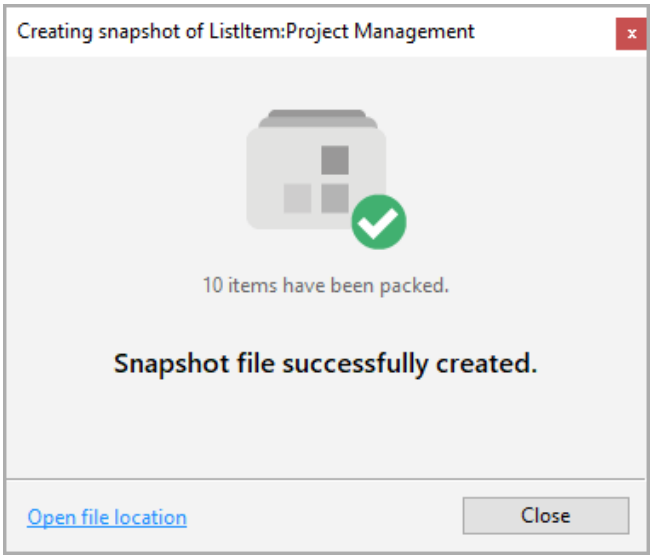
Save the snapshot file

Finally, you need to save the **Scroll** -created snapshot from the working memory to permanent storage.

7. In the element tree toolbar**, click** the **Save as snapshot…** button.
8. **Name** the file, **specify** where it should be saved, and **click Save**.



9. A message shows that the snapshot file was saved successfully.

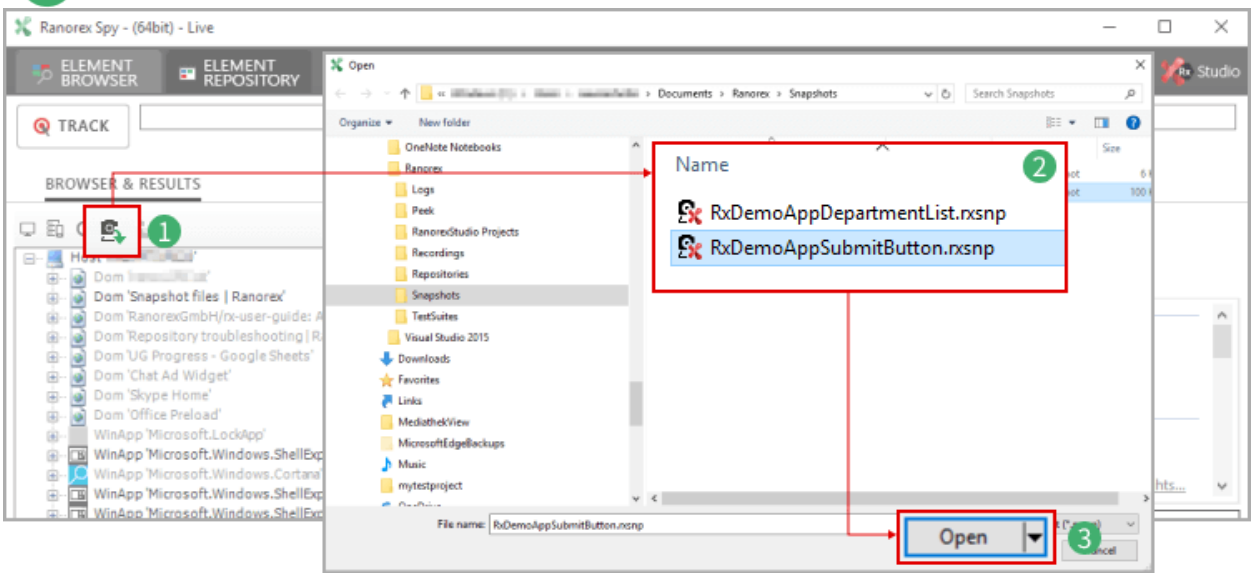


Load a snapshot file

You can also load snapshot files in Ranorex Spy.

To do so:

1. **Start** Ranorex Spy and in the element tree toolbar, **click** the **Load from snapshot…** button.
2. **Browse** to the snapshot file you want to open.
3. **Click Open.**

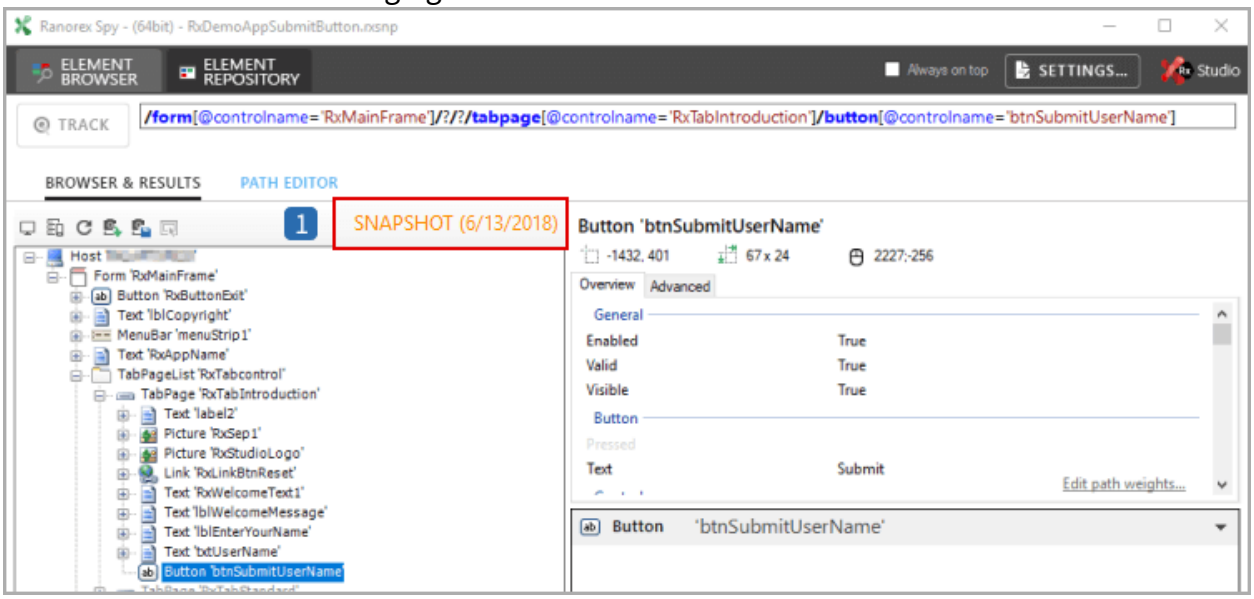


1. Start Ranorex Spy and click **Load from snapshot…** in the toolbar
2. Choose **folder** and saved **snapshot** file

3. Click **Open**

> **Note**
>
> The default save location for snapshot files is `%USERPROFILE%\RanorexSnapshots`.

4. The snapshot file is opened in Spy, as indicated by the new element tree and the status indicator changing from LIVE to SNAPSHOT.



1. Status indicator showing the snapshot file's date of creation.

GDI capture feature

In rare cases, Ranorex Studio can't identify UI elements of an AUT correctly. This applies to certain technologies such as VB6, MFC, and older Delphi versions. In these cases, you can use the GDI (graphic device interface) capture feature to identify these elements correctly. Ranorex Studio supports both GDI and GDI+. You have to enable GDI+ recognition manually. This is explained at the end of this page.

The GDI capture feature is part of Ranorex Spy. You can access it from an element's context menu in the element tree browser.

On this page, we'll explain the difference between non-GDI and GDI UI-element identification by way of a simple example in the Ranorex Studio Demo Application: the calendar.

Screencast
The screencast "GDI capture feature" walks you through information found in this chapter:

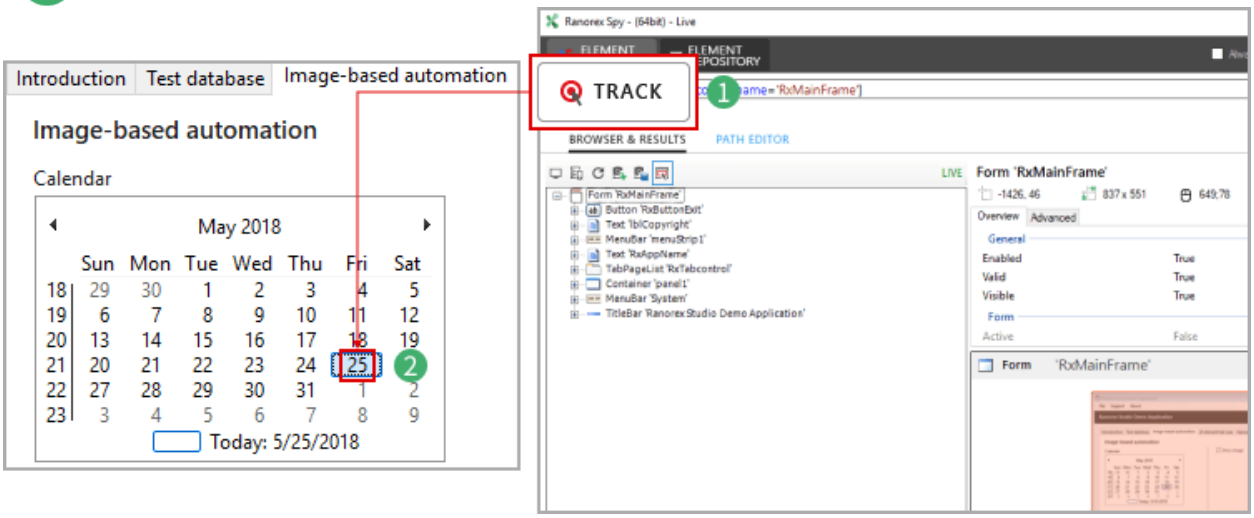
Watch the screencast now

Default, non-GDI UI-element identification

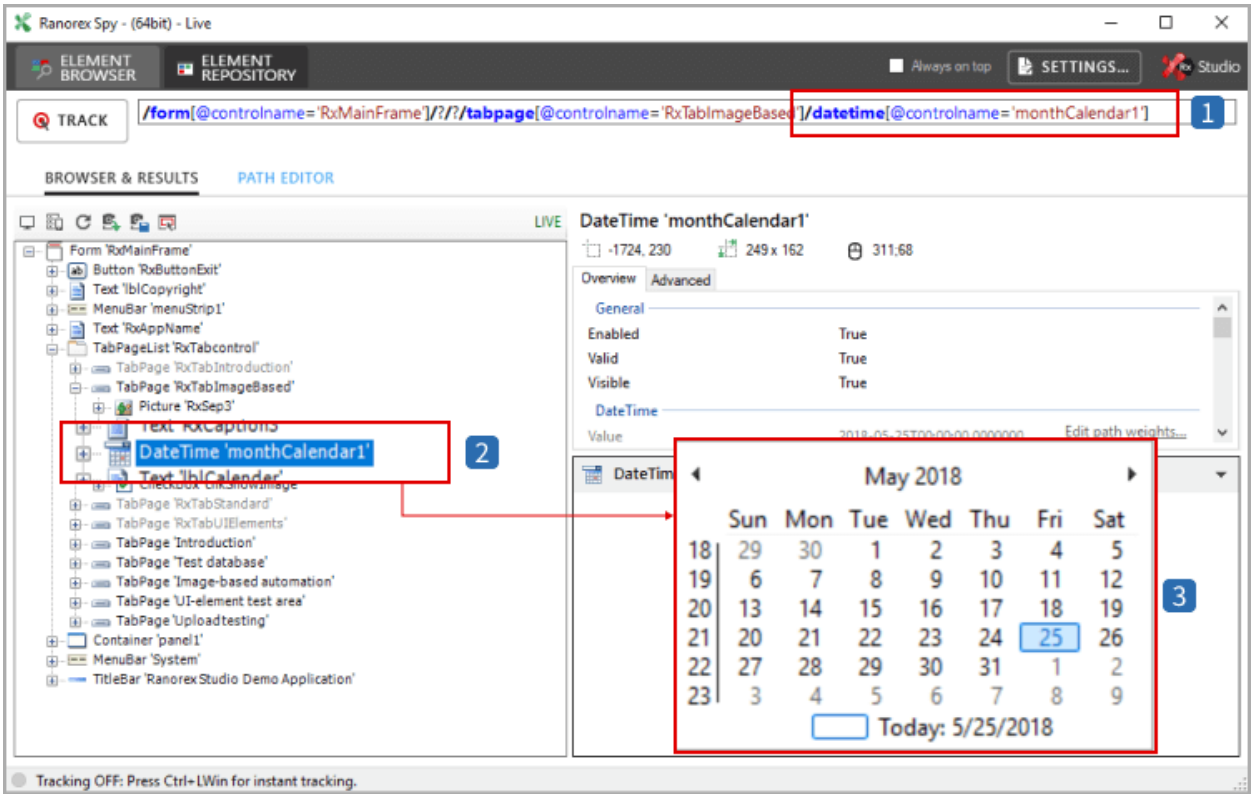
We want to automate the calendar in the Demo Application. For this purpose, we need to identify the UI element for a specific day of a month.

Let's identify the element by using the track function in Spy:

1. **Start** Ranorex Spy and click **TRACK**.
2. In the Demo Application, click a specific day, e.g. 25 May.



Result

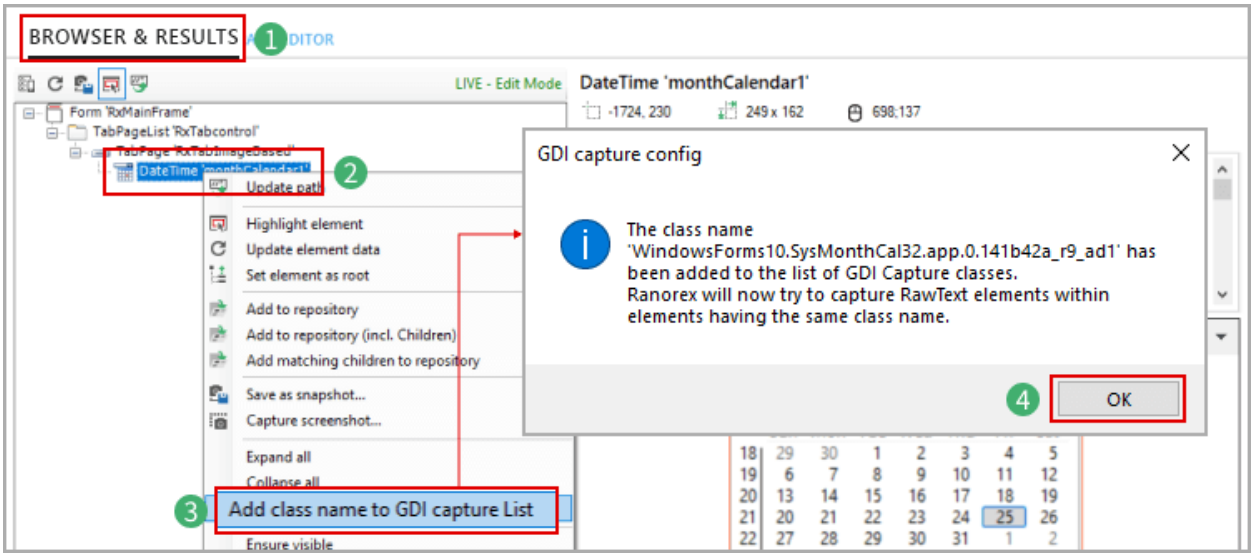


1. The RanoreXPath of the new UI element shows that not a specific day, but the entire calendar has been identified.
2. In the element tree, the UI-element also has the adapter type `DateTime` (i.e. a calendar).
3. The screenshot for the UI element also shows the entire calendar, and not just the 25.

Enable GDI capture

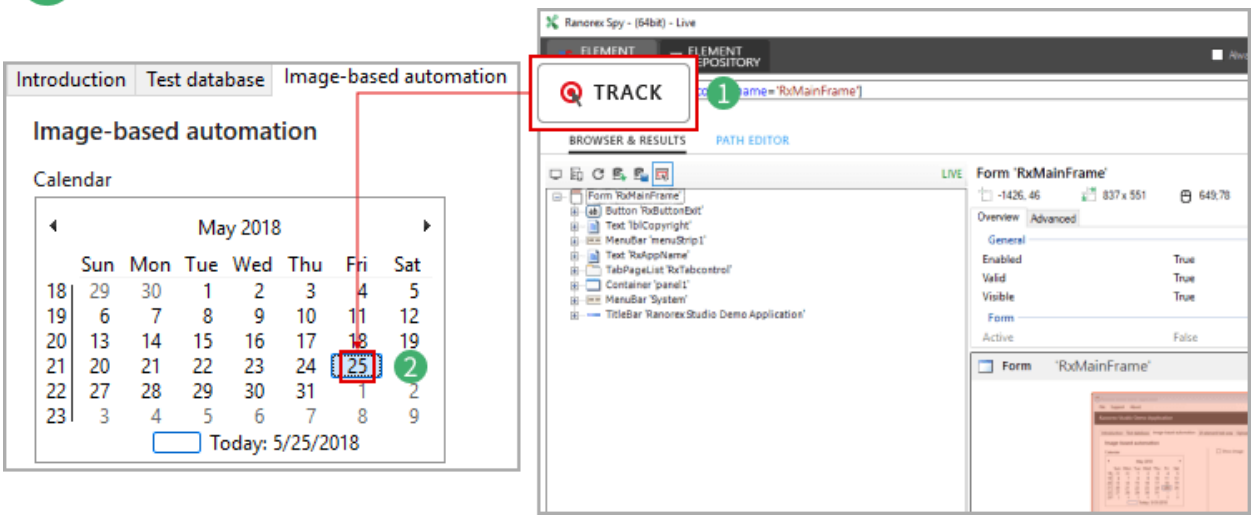
As default UI element identification didn't work in this case, we need to enable GDI capture. To do so, we add the incorrect UI element to the **GDI capture list.** In our case, this will allow Ranorex Studio to identify the correct UI element based on the *RawText* adapter class.

1. **In the element tree, select** the **UI-element** you want to be handled by GDI capture.
2. Right-click it and click **Add class name to GDI capture list**.
3. **Click OK** to confirm the info dialog



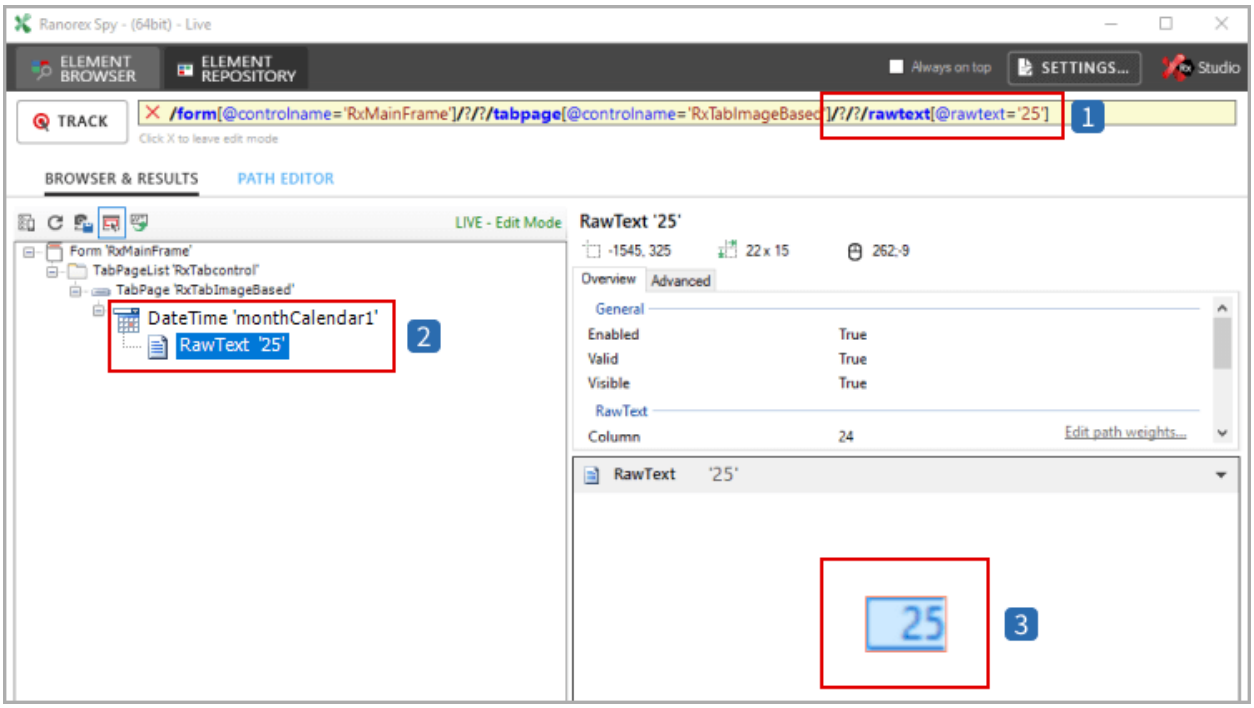
Now we can track the UI element once more.

1. **Click TRACK.**
2. **Click** a specific day in the calendar, e.g. 25 May.



Result

The UI element is identified correctly based on the *RawText* adapter.



1. The RanoreXPath shows that the date a specific day, and not the entire calendar, has been identified.
2. In the element tree, the UI element has the adapter type `RawText`.
3. The screenshot now shows the UI element for the specific day instead of the entire calendar.

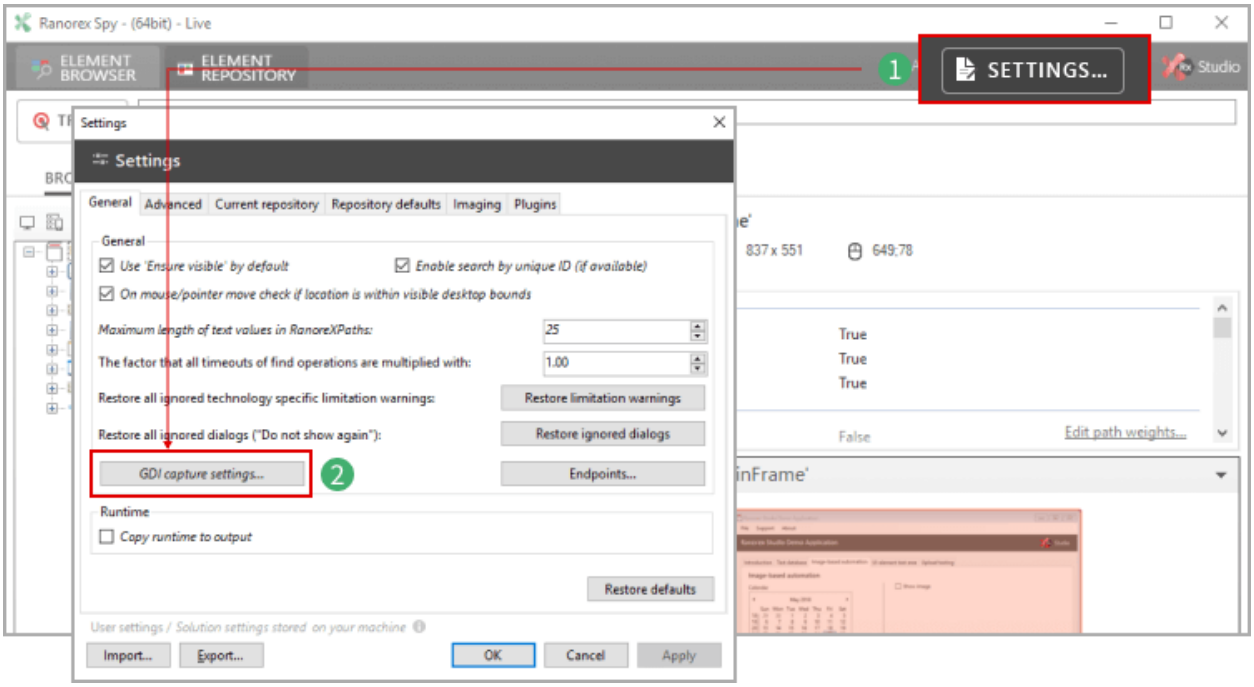
Add a process to the GDI capture list

We recommend only adding single UI element classes to the GDI capture list. However, if you need to add an entire process (e.g. for legacy reasons), you can do so in the GDI capture settings, which is explained in the next topic.

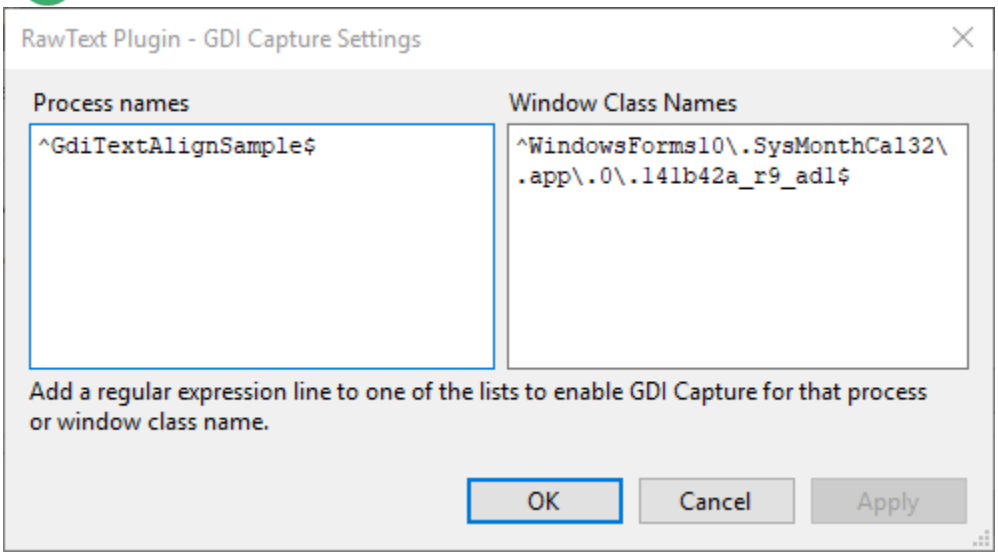
GDI capture settings

The GDI capture settings display all added processes and class names. This is also where you can add processes and class names in bulk or by specifying a regular expression. You can access the GDI capture settings from the Settings dialog in Ranorex Studio or Spy.

1. **Click SETTINGS...**.
2. In the **General** tab, **click GDI capture settings...**.

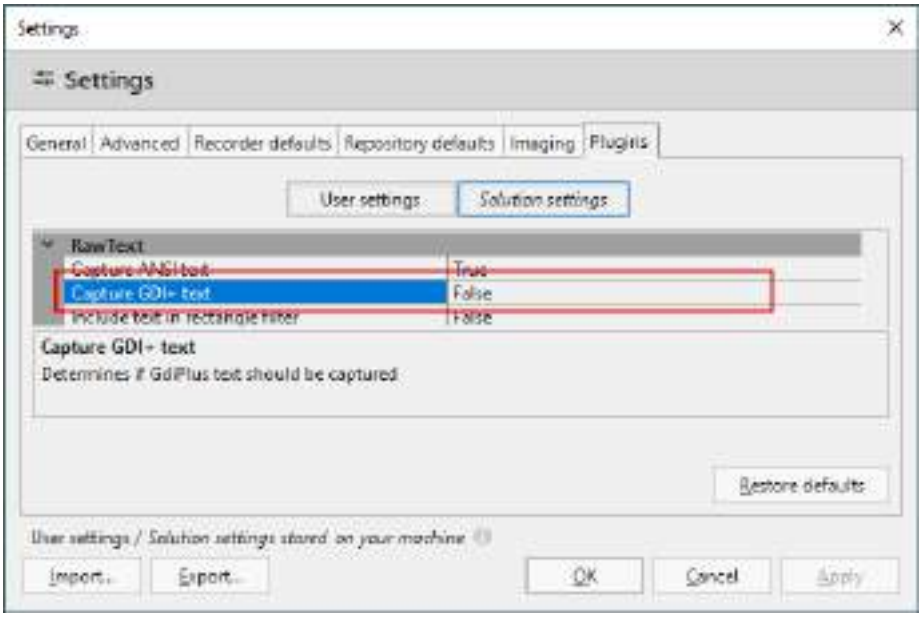


3 The GDI settings open.



Enable GDI+

GDI+ recognition is turned off by default. If required, enable it manually under **Settings > Plugins > Capture GDI+ text.**


Settings
Settings
General
Advanced
Recorder defaults
Repository defaults
Imaging
Plugins
User settings
Solution settings
RawText
Capture ANSI text
True
Capture GDI+ text
False
Include text in rectangle filter
False
Capture GDI+ text
Determines if GdiPlus text should be captured
Restore defaults
User settings / Solution settings stored on your machine
Import...
Export...
OK
Cancel
Apply

UI elements

Graphical user interface elements (i.e. UI elements) are the graphical elements on a computer screen that represent stored information within computers and allow users to interact with the software. Common examples are windows, text fields, buttons, labels, lists, selection elements.

Because Ranorex Studio is a tool for automated GUI testing, it needs to be able to reliably identify UI elements for use in the tests you create. In this chapter, we'll explain how this works.

Screencast

The screencast "Roles and capabilities"walks you through the information found in this chapter:

Watch the screencast now

This chapter covers an advanced topic that is closely connected to Ranorex Spy and RanoreXPath. Together, they explain how UI element identification in Ranorex Studio works. We therefore recommend you also study these chapters.

Further reading

RanoreXPath is explained in: Ranorex Studio advanced > ⇢ RanoreXPath.

Further reading

Ranorex Spy is explained in: Ranorex Studio advanced > ⇢ Ranorex Spy.

UI elements

Graphical user interface elements (i.e. UI elements) are the graphical elements on a computer screen that represent stored information within computers and allow users to

interact with the software. Common examples are windows, text fields, buttons, labels, lists, selection elements.

Because Ranorex Studio is a tool for automated GUI testing, it needs to be able to reliably identify UI elements for use in the tests you create. In this chapter, we'll explain how this works.

Screencast

The screencast "Roles and capabilities"walks you through the information found in this chapter:

Watch the screencast now

Related chapters

This chapter covers an advanced topic that is closely connected to Ranorex Spy and RanoreXPath. Together, they explain how UI element identification in Ranorex Studio works. We therefore recommend you also study these chapters.

Further reading

RanoreXPath is explained in:

Ranorex Studio advanced > ⇢ RanoreXPath.

Further reading

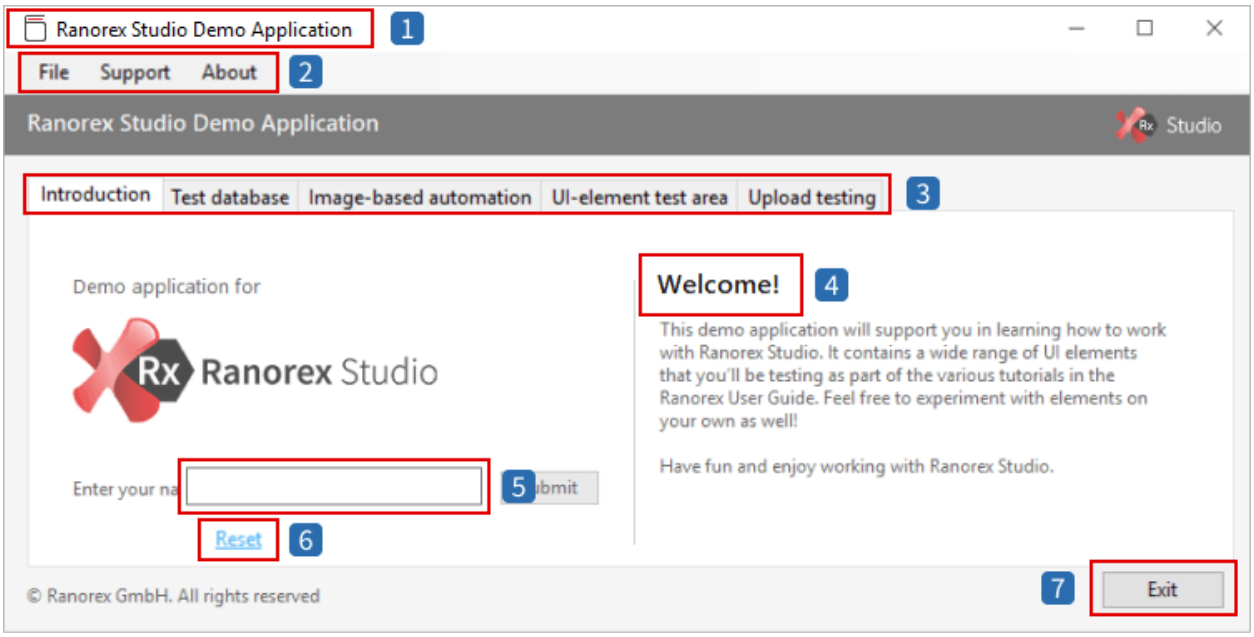
Ranorex Spy is explained in:

Ranorex Studio advanced > ⇢ Ranorex Spy.

UI element identification

Let's first take a look at how we as humans identify UI elements. For us, the process is usually quite intuitive.

Take the start page of the Ranorex Studio demo application, for example:



1. **Title bar**
2. **Main menu**
3. **Tabs for different panels**
4. **Text label** showing a welcome message
5. **Text field** for entering a name
6. **Text link** for resetting the welcome message
7. **Exit button** for closing the application

User's point of view

To better understand the challenge of reliable UI element identification, let's take a look at one particular UI element, the text input field.

How do you know that the framed UI element is a text input field?

Possible answers may be:

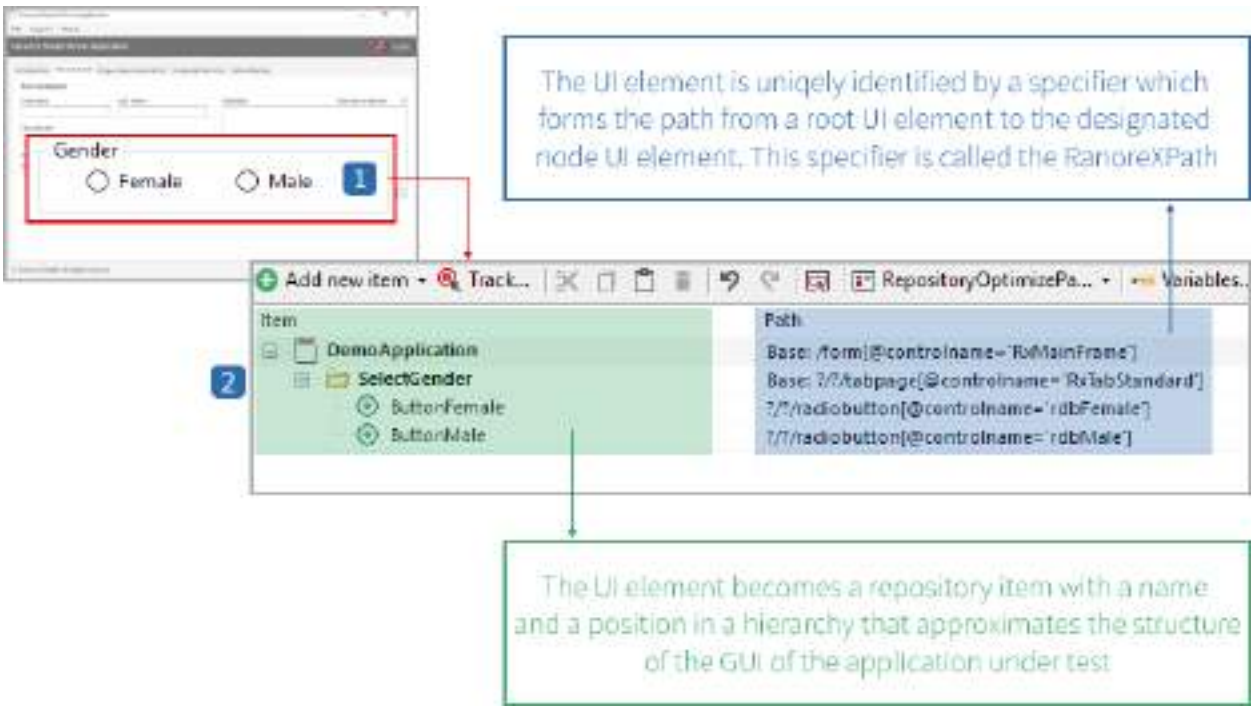
- I read it in the user guide of the software
- I know from experience
- I tried entering text and it worked
- Someone told me/showed me
- Because of the prompt ‘Enter your name’ and the ‘Submit’ button

All of these are good indicators, and if the field accepts text entry, you can be quite certain that it’s a text input field. However, unless you have access to the software’s code, you can’t tell with absolute certainty what the UI element does in detail. Instead, you have to rely on context-based clues and your general idea of what a text input field does, i.e. an “inner representation” of the UI element.

Ranorex Studio’s approach

As an application, Ranorex Studio doesn’t have access to indicators like user guide information, being shown by others, or human intuition. Its approach is based on hooking into an AUT, detecting which technology is used for the GUI, and then identifying UI elements and categorizing them according to ⇢ roles, capabilities, etc.

However, in some ways, Ranorex Studio works in a very similar way to human UI element identification: It also generates “inner representations” of UI elements, called repository items. These have a unique identifier assigned to them, the RanoreXPath, that locates the respective UI element in the GUI structure of the software. This is a two-step process:



1 **UI elements** in the GUI of the AUT

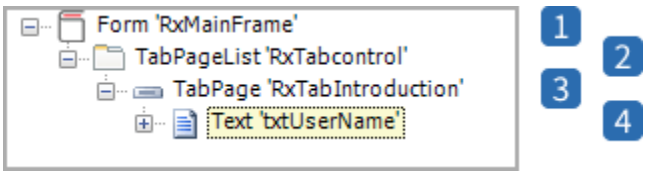
2 **Representation** of these UI elements

Important to know:

- Ranorex Studio represents UI elements as repository items in the test solution.
- Ranorex Studio identifies the type of the UI element on a best-guess method and maps it to a set of specific roles and capabilities.
- Ranorex Studio assigns a unique identifier to every repository item. This identifier is called an item's RanoreXPath and reflects the position of the referenced UI element in the GUI structure.

GUI structure

The UI elements in an AUT are organized in a certain structure. When Ranorex Studio scans a GUI to identify UI elements in it, it translates this structure into a hierarchical element tree that you can see in ⇢ Ranorex Spy. The animation below illustrates this:



1 Form `RxMainFrame`

- The top-level UI element of the AUT, i.e. the demo application program window as a whole.
- This UI element contains almost all other UI elements of the demo application (there are special cases like context menus and list items, which normally appear as separate branches in the element tree).
- Examples for UI elements contained in this tree are the **Exit** button, the **main menu**, or the **copyright text label.**

2 TabPageList `RxTabControl`

- The tab container with the tabs Introduction, Test database, Image-based automation, UI-element test area, and Upload.
- Ranorex Studio gets the name of this UI element (RxTabControl) directly from the AUT.

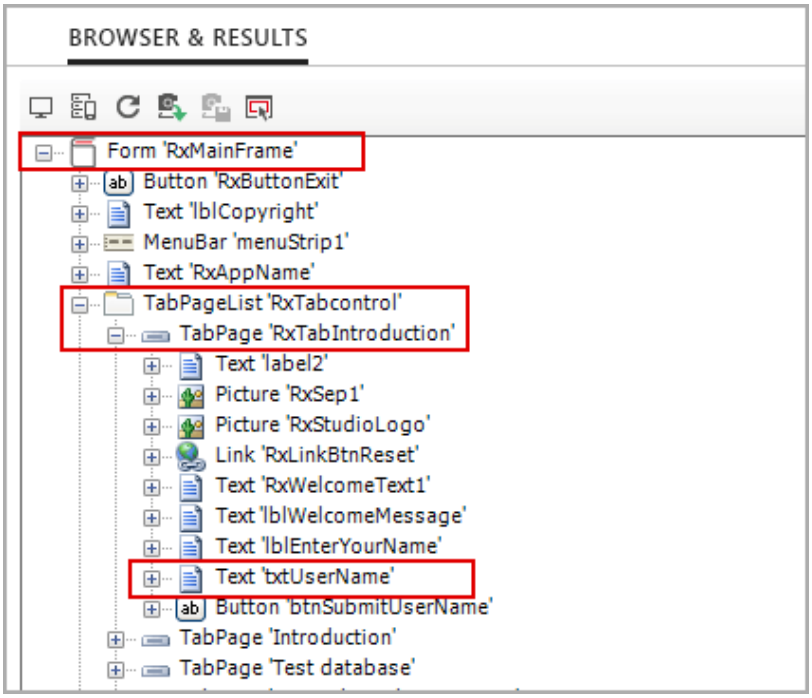
3 TabPage `RxTabIntroduction`

- The Introduction tab in the demo application.
- This UI element contains all other elements present in this tab.
- Ranorex Studio gets the name of this UI element (RxTabIntroduction) directly from the AUT.

4 Text field `txtUsername`

- The text input field for changing the name that's displayed in the welcome message.
- This UI element, along with the others in this panel, is contained in the tab element RxTabIntroduction.

The above example is simplified for explanation purposes. The full hierarchical tree would look like this in Ranorex Spy:



Roles, capabilities, and more

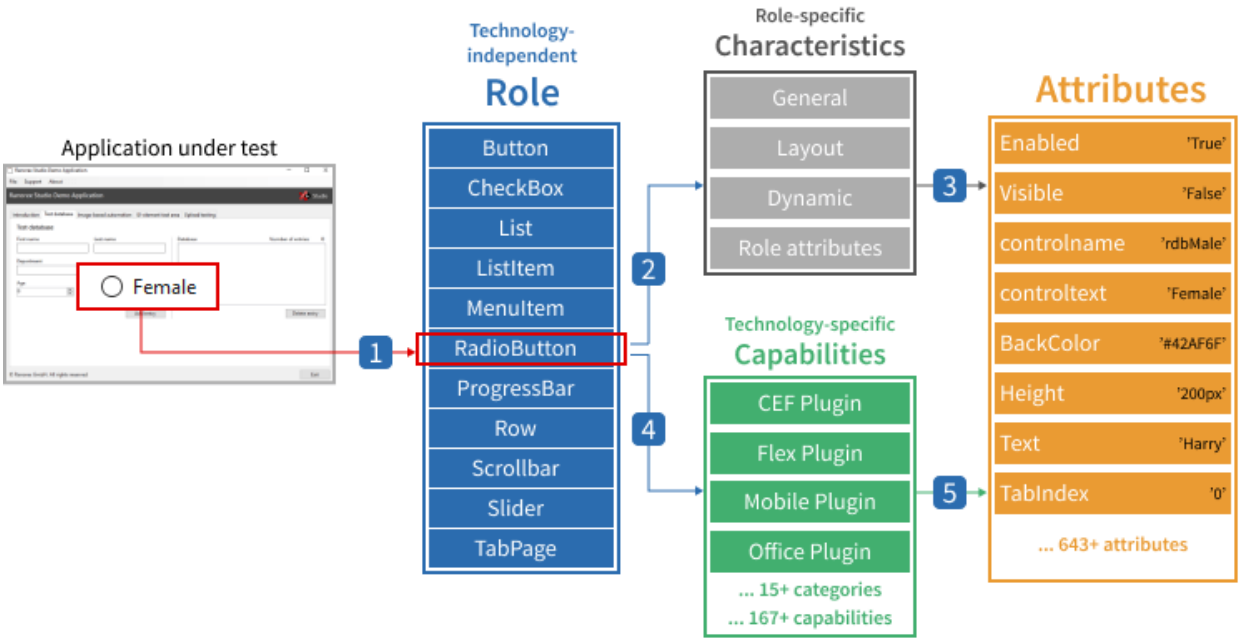
When tracking and identifying UI elements, Ranorex Studio also categorizes them according to their purpose and to define their states. The categories Ranorex Studio uses are roles, capabilities, characteristics, and attributes.

Ranorex Studio chooses these categories based on a best-guess method. If it can't categorize an UI element, it classifies it as *Unknown*.

This page explains how the categorization process works, which roles, capabilities, and characteristics there are, and where in Ranorex Studio you can find them.

UI element categorization process

Ranorex Studio follows a certain order when categorizing UI elements and assigning roles, capabilities, etc. The image below shows this:



1 Technology-independent role assignment

- Ranorex Studio first defines the role of the UI element.
- The role defines how a user can interact with the UI element.
- If Ranorex Studio can't find an adequate role, it's set to **Unknown.**

2 Role-specific characteristics

- Based on the role, Ranorex Studio derives a set of characteristics.
- These characteristics contain sets of attributes (see item 3) to define the UI element in more detail.

3 Attributes

- Each characteristic comes with a set of attributes.
- Attributes take values to define the UI element in more detail.
- Together with those derived from the capability category (item 4), there are more than 600 attributes.

4 Technology-specific capabilities

- Developers use one or more technologies to build the UI of their application.
- These technologies contain different capabilities, which define what a UI element can do.
- Based on the technology used for the UI element, Ranorex Studio derives a set of capabilities.

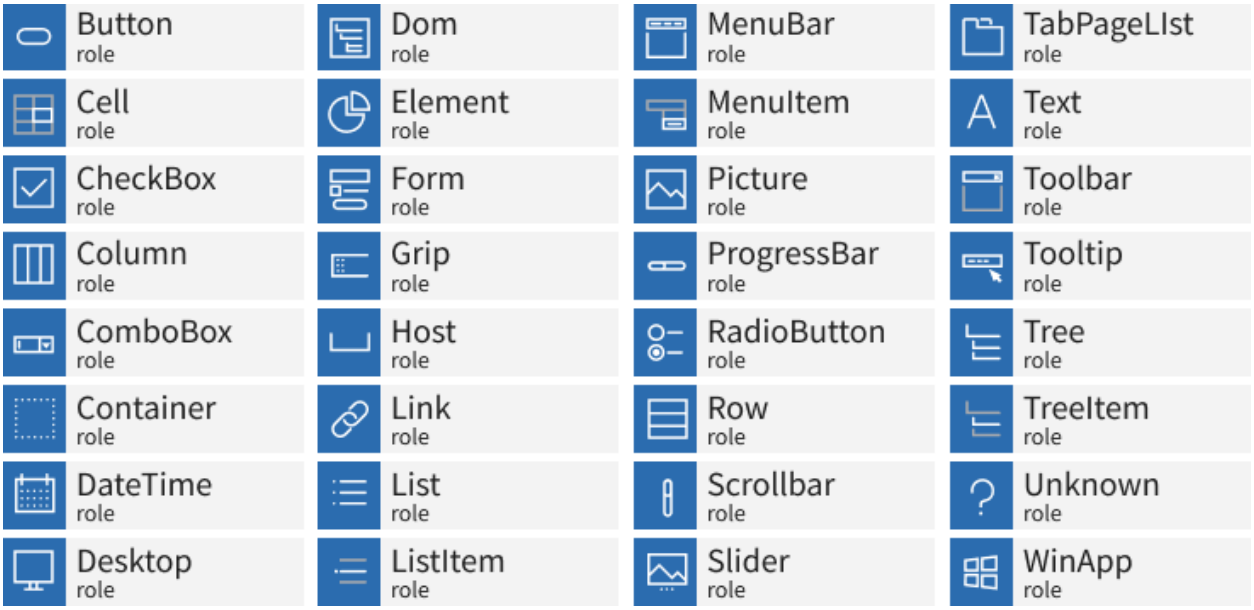
- Capabilities contain sets of attributes (see item 5) to define the UI element in more detail.
- There are around 150 capabilities divided into 15 categories.
- As technology evolves, some capabilities become outdated and new ones are be added.

5 Attributes

- Each capability comes with a set of attributes.
- Attributes take values to define the UI element in more detail.
- Together with those derived from the characteristic's category (item 3), there are more than 600 attributes.

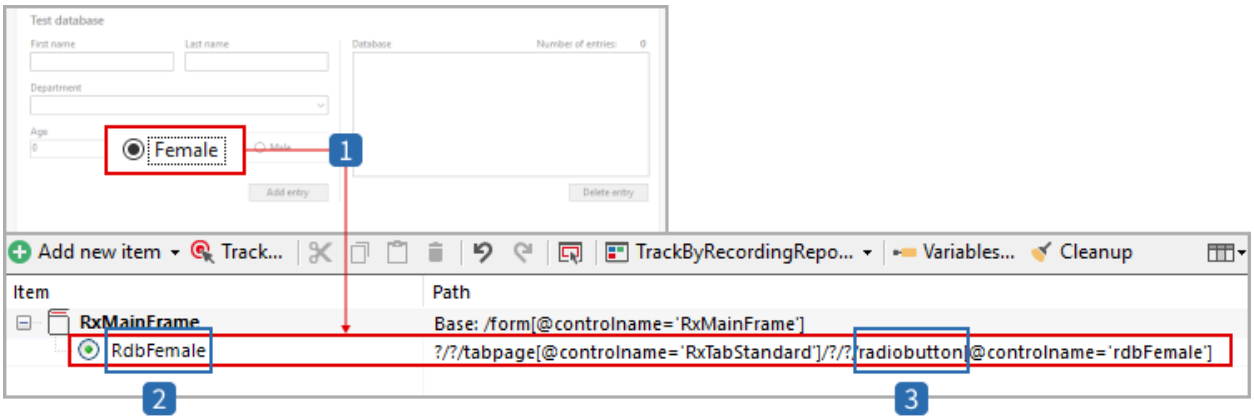
Roles

Roles define how a user can interact with an UI element, i.e. the UI element's purpose. Below you can see a table of predefined roles in Ranorex Studio. These are technology unspecific and cover the most common UI elements. The Unknown type is for when Ranorex Studio can't assign an adequate role based on its algorithm.

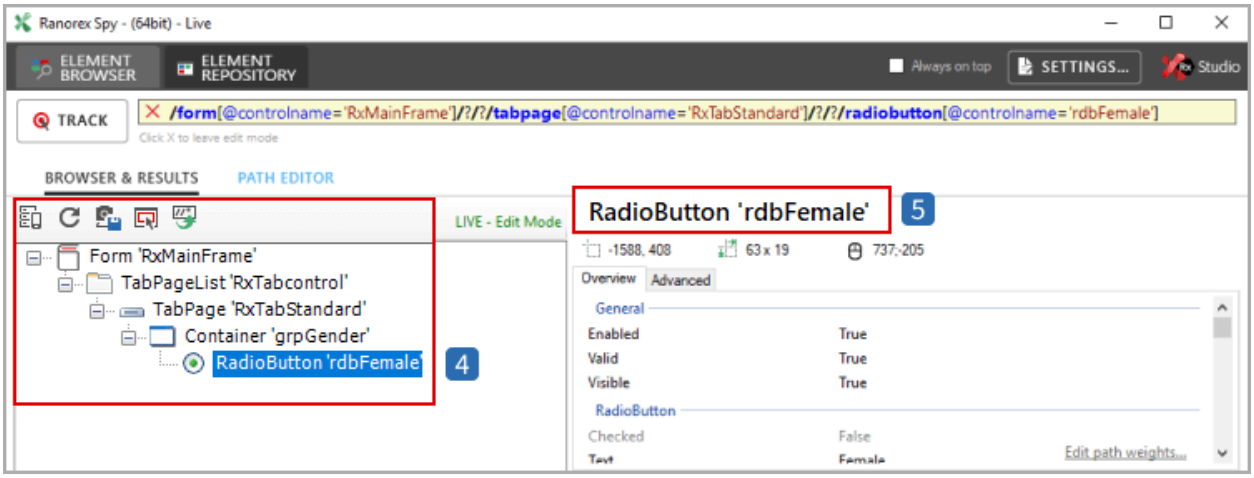


Example for role assignment

For a quick example of an assigned role, simply take a look at a repository item's RanoreXPath or select a UI element in the tree browser in Ranorex Spy.



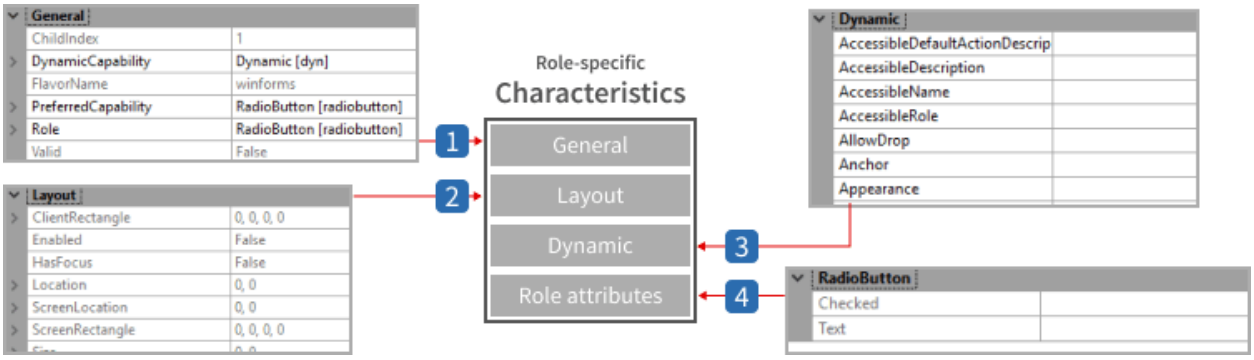
1. **Radio button** for gender selection in AUT.
2. Corresponding **repository item** named `RdbFemale` in the repository.
3. Assigned **role** in the item's RanoreXPath.



4. Spy displaying the UI element with the assigned role outside quotes in the UI element name (element tree)
5. Spy displaying the UI element with the assigned role outside quotes in the UI element name (element details)

Role-specific characteristics

Based on the role, Ranorex Studio derives a set of characteristics. They in turn contain sets of attributes that define the UI element in more detail. The number and type of characteristics varies depending on the role. The following image shows an example for a set of characteristics for the role Radio button.



1 General

- Attributes and actions in this category are directly derived from the UI element.
- This category sums up attributes and actions which may originate from other categories.

2 Layout

- Contains all attributes that refer to the graphical layout of the UI element.
- Attributes in Layout are technology independent.

3 Dynamic

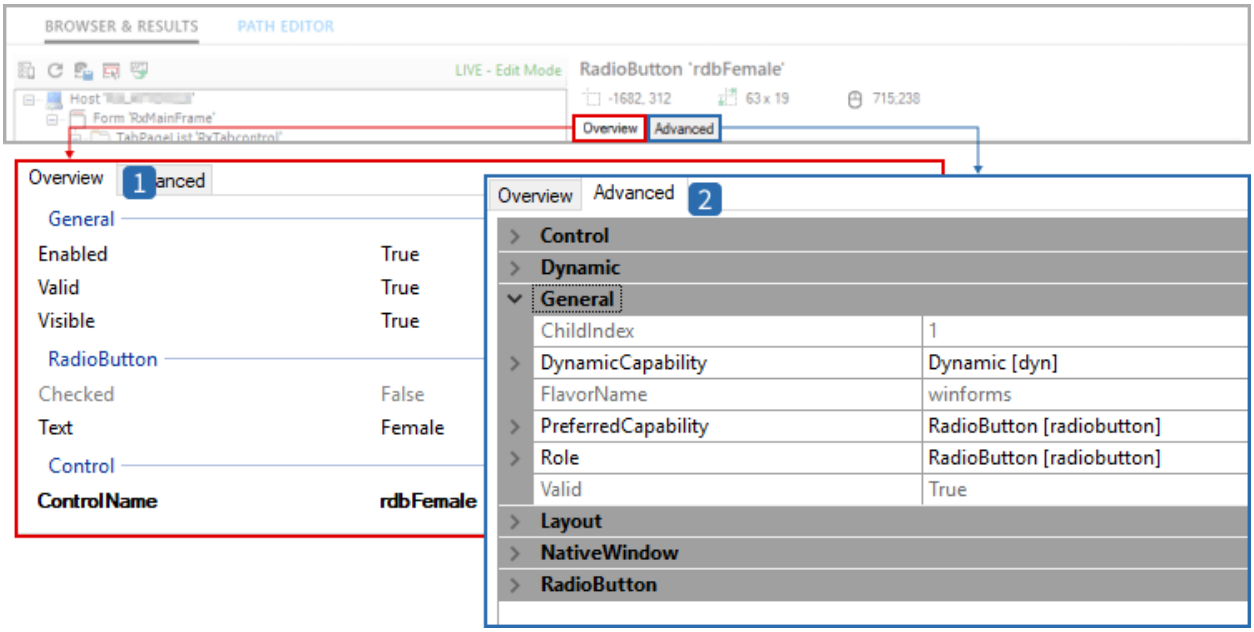
- When a UI element has been implemented with extended functionality by the developer, the corresponding attributes are contained in this category.

4 Role attributes

- Contains attributes that only apply to this specific role.
- Takes the name of the role. In the example, the characteristic is called `RadioButton`.

View role-specific characteristics

You can view characteristics and their attributes in Ranorex Spy under the Overview and Advanced tabs in the element details.



1 Attributes **overview**: shows a summary of the currently available and assigned attributes of the UI element.

2 Attributes **advanced** view: shows the attributes in detail and lets you edit them.

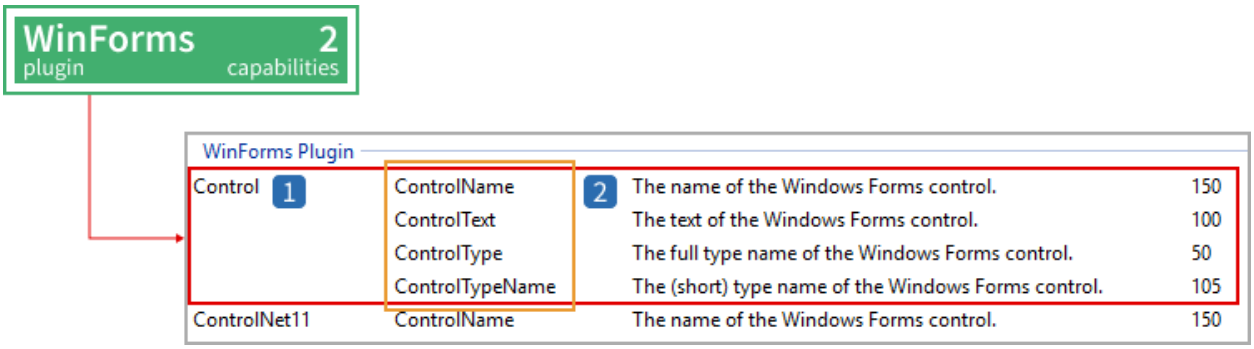
Technology-specific capabilities

Aside from the role-specific characteristics above, Ranorex Studio also assigns technology-specific capabilities that in turn contain sets of attributes. Capabilities are organized in technology categories. The table below shows these categories and the number of capabilities they contain.

CEF plugin — 2 capabilities	Office plugin — 8 capabilities	Web plugin — 116 capabilities
Flex plugin — 4 capabilities	Qt plugin — 3 capabilities	WebDriver plugin — 1 capability
Java plugin — 2 capabilities	RawText plugin — 3 capabilities	Win32 plugin — 3 capabilities
Mobile plugin — 11 capabilities	SAP plugin — 3 capabilities	WinForms plugin — 2 capabilities
MSAA plugin — 1 capability	UIA plugin — 5 capabilities	WPF plugin — 3 capability

Example for capability assignment

Let's use the radio button form our previous examples here again. The image below shows the attributes overview for the radio button in Ranorex Spy. We can see that it has been assigned the `Control` capability, which contains four attributes and is part of the WinForms technology. This is because the AUT uses the WinForms technology to implement the radio button.

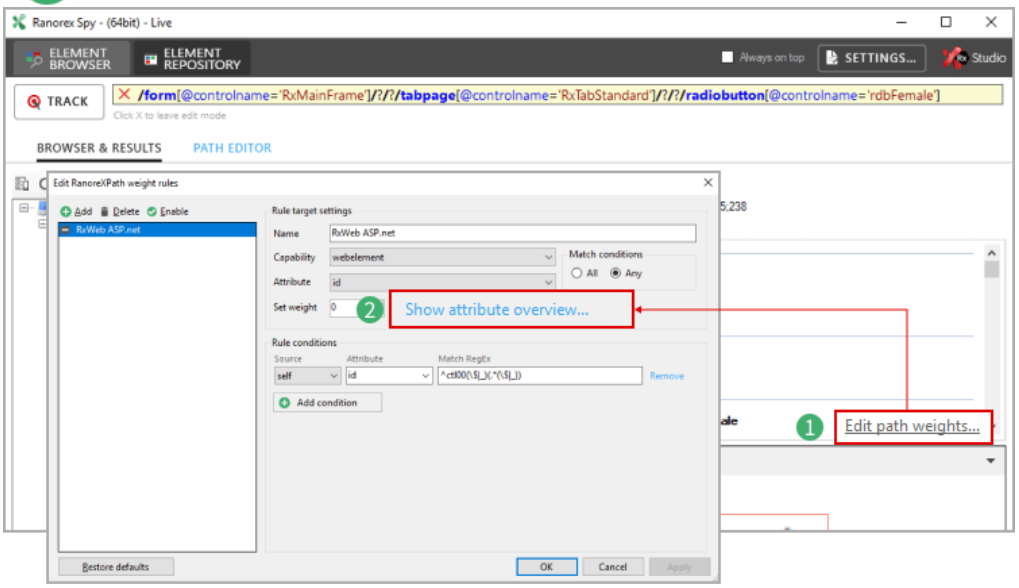


1. The Control capability, part of the WinForms technology.
2. The four attributes this capability contains.

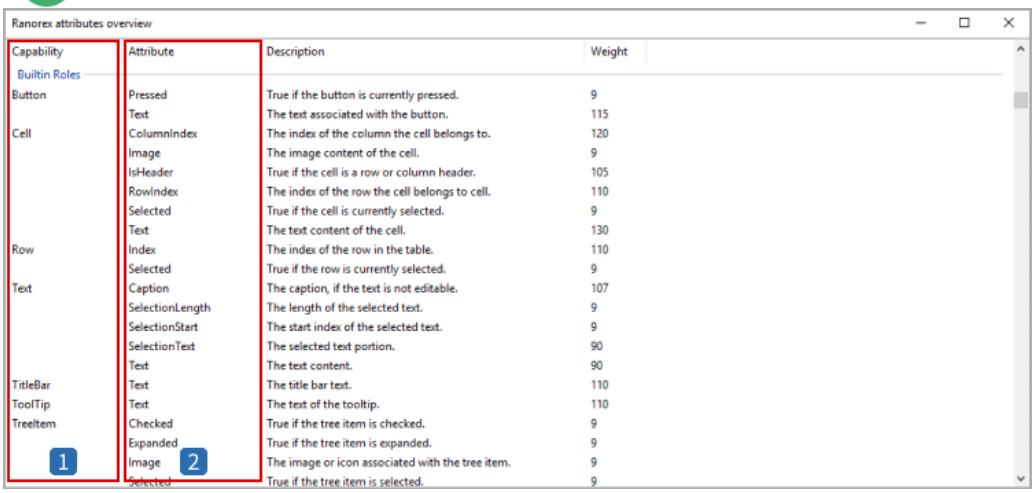
View all capabilities and their attributes

You can view all capabilities and attributes through the **Edit path weights…** menu in Ranorex Spy.

1. In the attributes overview in Spy, **click Edit path weights…**.
2. In the displayed dialog, **click Show attribute overview…**.



3 The attributes overview opens.



1 **Capabilities**

2 **Attributes** for each capability

RanoreXPath

The purpose of RanoreXPath expressions is to uniquely identify UI elements in a desktop, web, or mobile application. Without RanoreXPath, Ranorex Studio couldn't find the UI elements it's supposed to perform actions on.

The RanoreXPath syntax is based on the XML description syntax XPath. The two are not the same, however.

The primary tool for working with RanoreXPath expressions is Ranorex Spy.

Recommended knowledge

To make the most out of the information in this chapter, we recommend you also study the following chapters:

Reference

Ranorex Spy is the primary tool for working with RanoreXPath expressions. It is explained in Ranorex Studio advanced > ⇢ Ranorex Spy

Reference

UI elements, roles, capabilities, and attributes are explained in Ranorex Studio advanced > ⇢ UI elements.

Expert topics

While primarily an advanced topic, there are also some expert chapters related to RanoreXPath. Progress to them once you've finished studying this chapter.

Reference

Identifying dynamic UI elements and using RanoreXPath weights are explained in

Ranorex Studio expert > ⇢ Identifying dynamic UI elements

Reference

Using regular expressions in Ranorex Studio is explained in Ranorex Studio expert > ⇢ Regular expressions in Ranorex Studio

RanoreXPath basics

On this page, you'll find out how RanoreXPaths are structured and how RanoreXPath characteristics relate to automated software testing. We'll first explain the RanoreXPath structure for an isolated UI element, and then for this UI element as part of the UI it is situated in. Finally, we'll explain what robustness and flexibility are in relation to RanoreXPaths and how to make use of them for test automation purposes.

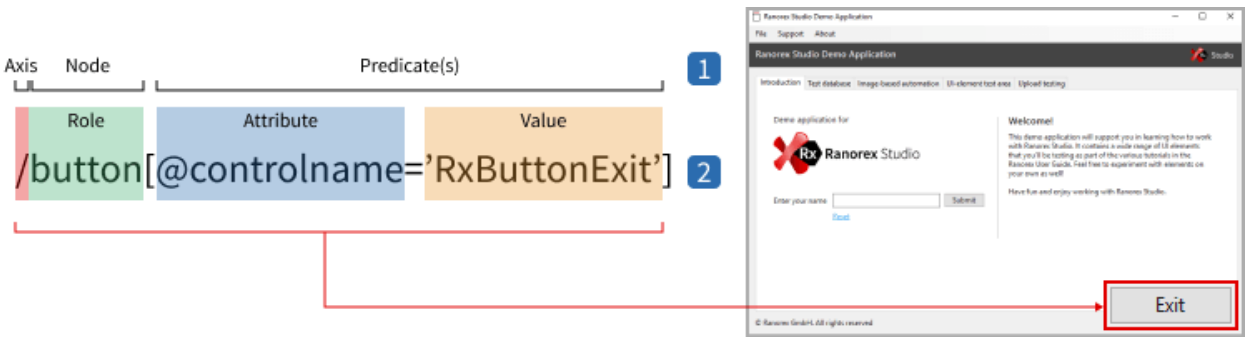
Screencast

The screencast "RanoreXPath blueprint" walks you through the information found in this chapter:

Watch the screencast now

Basic structure for an isolated UI element

Normally, a full RanoreXPath points to a UI element in the UI it is situated in. However, to explain the basic structure of a RanoreXPath for a single UI element, let's take a look at what's required to point to the **Exit** button in the Ranorex Studio Demo Application in isolation:



1. **Basic structure**: The RanoreXPath for an isolated UI element consists of three elements: an **axis** specifier, a **node**, and zero or more **predicate(s)** (all explained below)**.**
2. For the isolated **Exit** button, the axis specifier is `/`, the node is the **role** `button`, and the predicate consists of an **attribute-value pair** that uniquely identifies the button.

Axis

- Ranorex Studio treats UIs as a hierarchical tree.
- Axis specifiers indicate the way the RanoreXPath navigates within this tree.
- Examples for axis specifiers are `/`, `//`, `..`, and `ancestor`. For more information on axes, refer to ⇢ RanoreXPath syntax.

Node/role

- Node refers to a distinct node (i.e. UI element) in the UI element tree.
- In RanoreXPath, the UI element's role is the primary property by which it is identified.
- For more detailed identification, roles can be further specified with predicates enclosed in square brackets `[]`.

Predicate/attribute-value pair

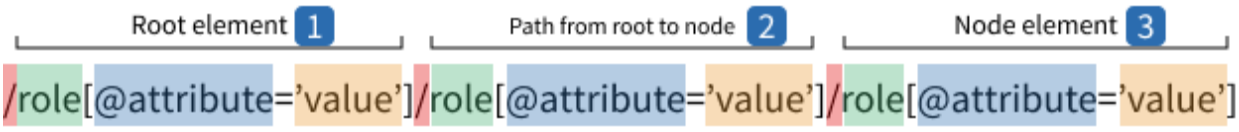
- Predicates further specify a node/role. They are optional.
- A predicate normally consists of one or more attribute-value pairs.
- Predicates are enclosed in square brackets.
- Attribute-value syntax is: `@attribute='value'`, where `=` is the operator.
- Operators other than `=` are available for use (explained ⇢ here).
- Predicates may also contain ⇢ regular expressions.

Structure of a full RanoreXPath

Above we explained the basic structure of the RanoreXPath for an isolated button, i.e. one isolated node in the UI element tree. We ignored the ancestor nodes that represent the UI this button is embedded in (e.g. the Ranorex Studio Demo Application window, the Introduction tab, etc.). Naturally, to identify the button in a real-life scenario, the RanoreXPath needs to navigate through all these ancestor nodes to get to the **Exit** button.

Chain isolated RanoreXPaths together

To get to the **Exit** button, we simply chain the RanoreXPaths for the isolated nodes/UI elements leading up to it together. This leaves us with the full RanoreXPath for the **Exit** button. This RanoreXPath consists of three parts that can be considered the basic parts of all RanoreXPaths:



1 **Root element**
The root of the current UI element tree. This is the starting point for navigating to the final node/UI element.

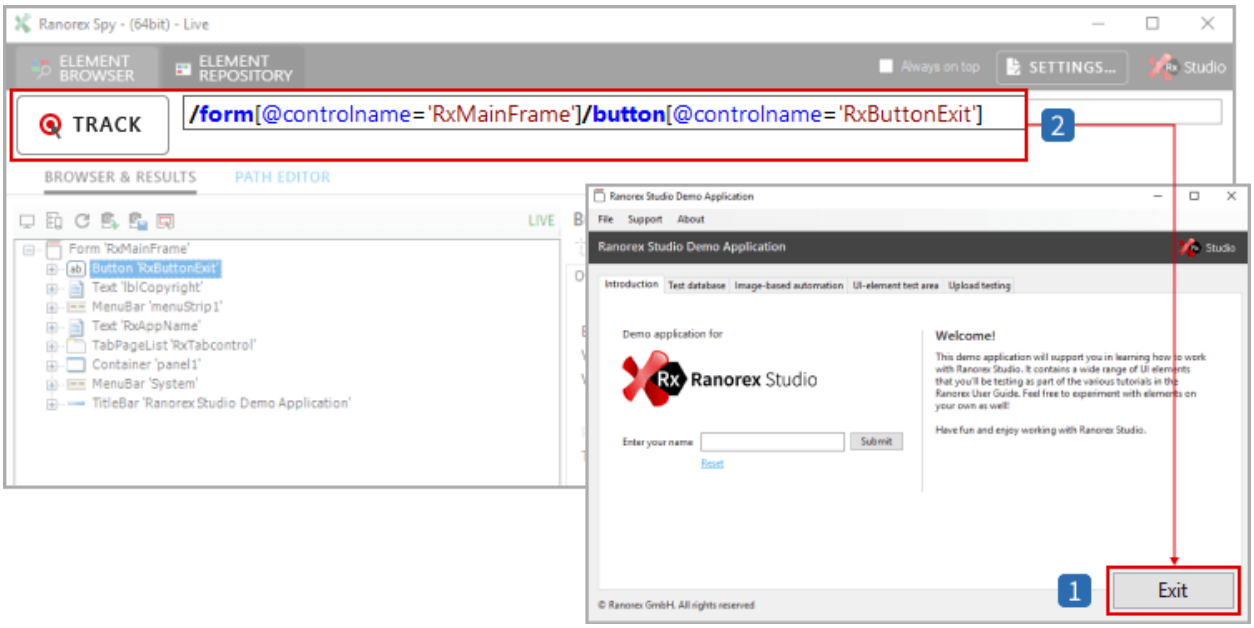
2 **Path from root to final node**
The nodes between root and final node. Can be zero or more.

3 **Final node/UI element**
The UI element you want to identify.

> **Note**
>
> All RanoreXPaths have these three parts, but these three parts don't have to be expressed in three nodes. They can also be expressed in just one node (e.g. when identifying only the Ranorex Studio Demo Application window), in two nodes (no other nodes between root and final node), or many more than three (several nodes between root and final node).

Example

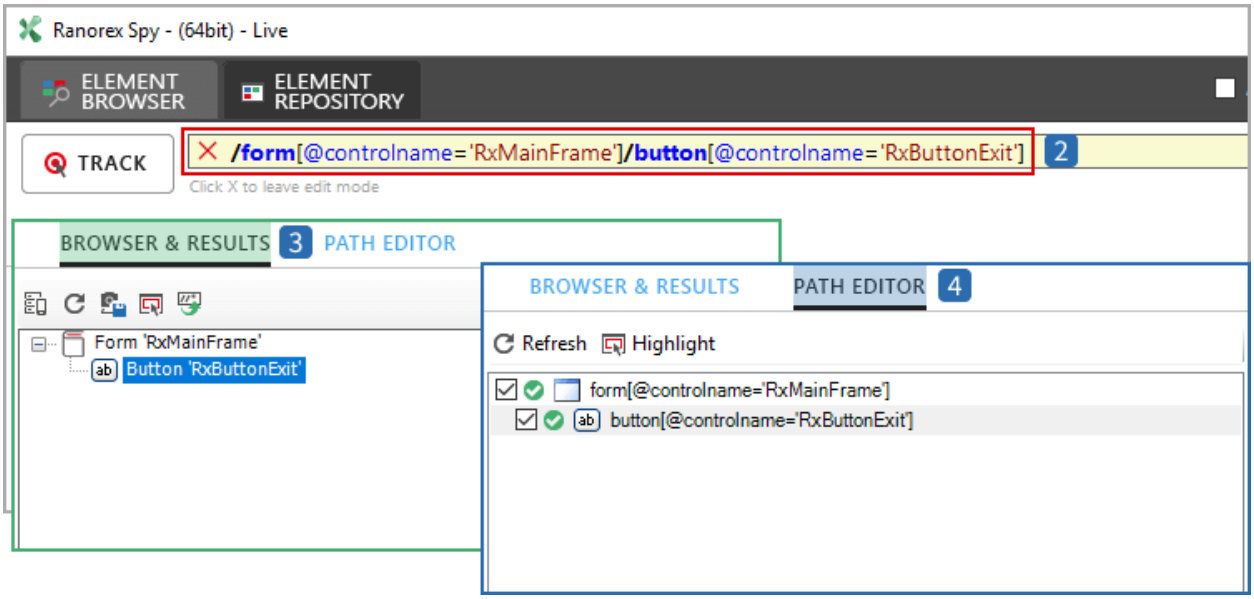
Let's take a look at the full RanoreXPath for the **Exit** button as it's generated when tracking the button from Ranorex Spy.



1 Tracked **Exit** button

2 The RanoreXPath generated for the button by Ranorex Spy.

The views in Spy provide more details and demonstrate which UI elements the RanoreXPath navigates through/identifies.

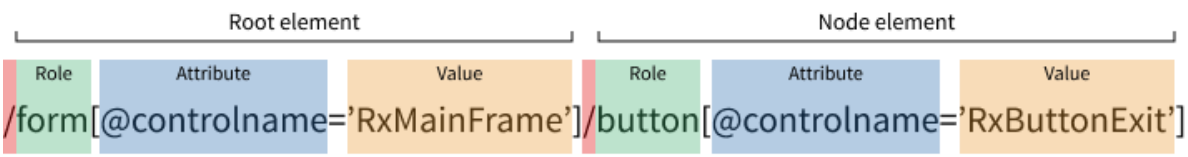


3 The **UI element tree browser** shows the two UI elements in the RanoreXPath in the UI hierarchy.

- The root element is the program window (role `form`) of the demo app.
- It is identified by the attribute-value pair `controlname='RxMainFrame'`.
- The final node is the **Exit** button.
- It is identified by attribute-value pair `controlname='RxButtonExit'`.
- There are **no** intermediate nodes between root and final node.

4 The **path editor** shows a more visual representation of the RanoreXPath, with details as to what node corresponds to what part of the RanoreXPath expression.

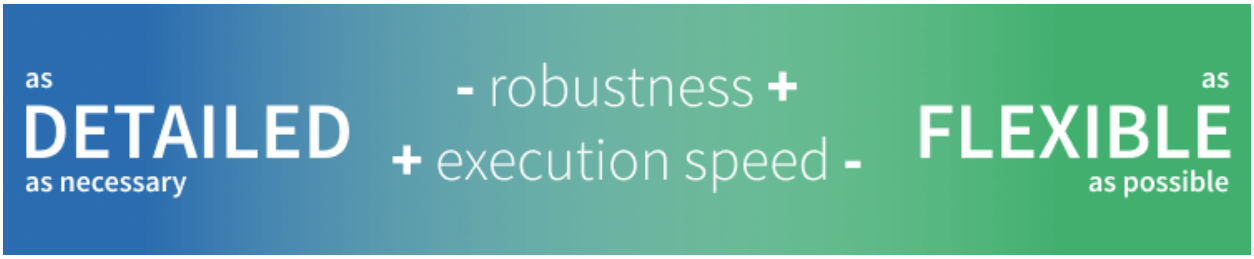
The full RanoreXPath expression for the **Exit** button is:



Robustness and flexibility

In UI test automation, robustness means that the test will still work if changes are made to the UI, e.g. moving a button from one place to another or changing its label. In Ranorex Studio, this is accomplished through the RanoreXPath.

An ideal RanoreXPath is as detailed as necessary (= faster UI-element identification and therefore execution speed) and as flexible (= more reliable UI-element identification, i.e. a more robust test) as possible.



- The level of detail vs. flexibility impacts robustness and execution speed.
- A very detailed RanoreXPath leads to less robust declarations but may speed up test execution.
- A very flexible RanoreXPath is more robust against UI changes, but may also slow down test execution.

Reference

Ranorex Studio generates RanoreXPaths based on a sophisticated algorithm that automatically balances robustness and flexibility. The behavior of this algorithm can be modified in Ranorex Studio under Settings > Advanced > RanoreXPath settings. This is explained in Ranorex Studio system details > Settings and configuration > ⇢ Advanced settings and configurations

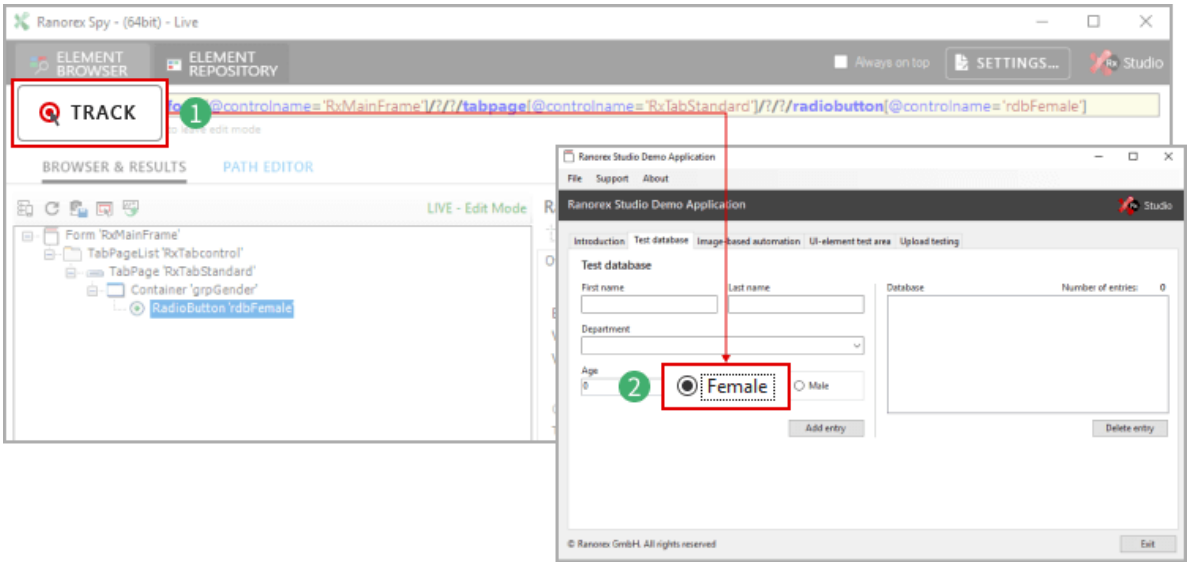
Increase flexibility with wildcards

When you want to increase the flexibility, and therefore the robustness, of a RanoreXPath, you can do so by adding wildcards to it. When Ranorex Studio generates RanoreXPaths automatically, it will also add wildcards if necessary, to balance the RanoreXPath between flexibility and UI element identification speed.

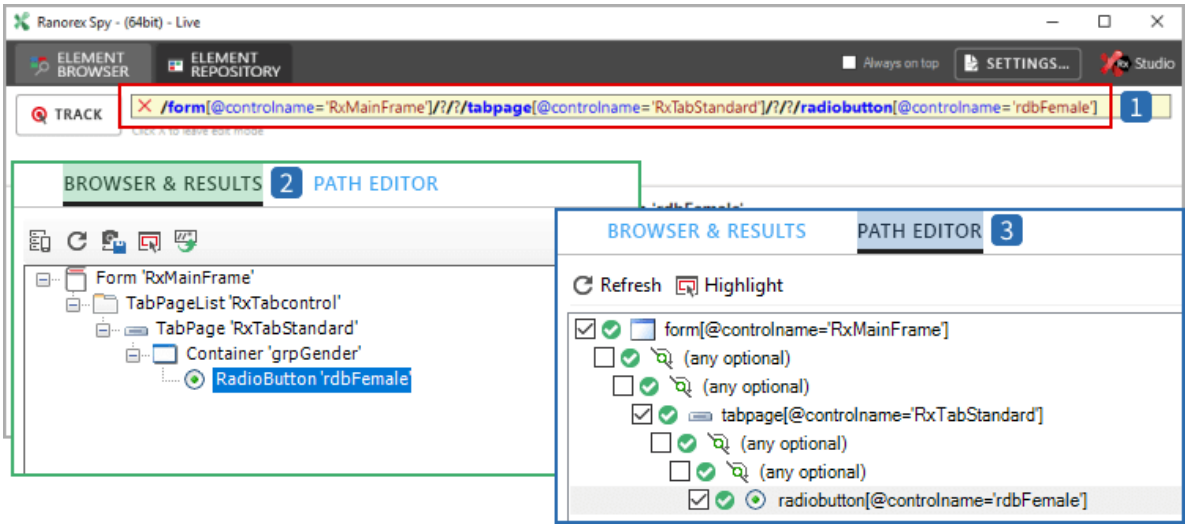
Example

Let's take a look at an example to see how Ranorex Studio automatically inserts wildcards. We'll track the Female radio button in the Test database tab of the Ranorex Studio Demo Application to have Ranorex Studio automatically generate a RanoreXPath:

1. In Ranorex Spy, **click Track**.
2. In the Ranorex Studio Demo Application, **click** the **Female** radio button in the **Test database** tab.



Ranorex Studio automatically generates a RanoreXPath for the UI element.



1. **RanoreXPath expression**:

- The generated RanoreXPath expression. Compare it to what's displayed in the element tree browser and the path editor.

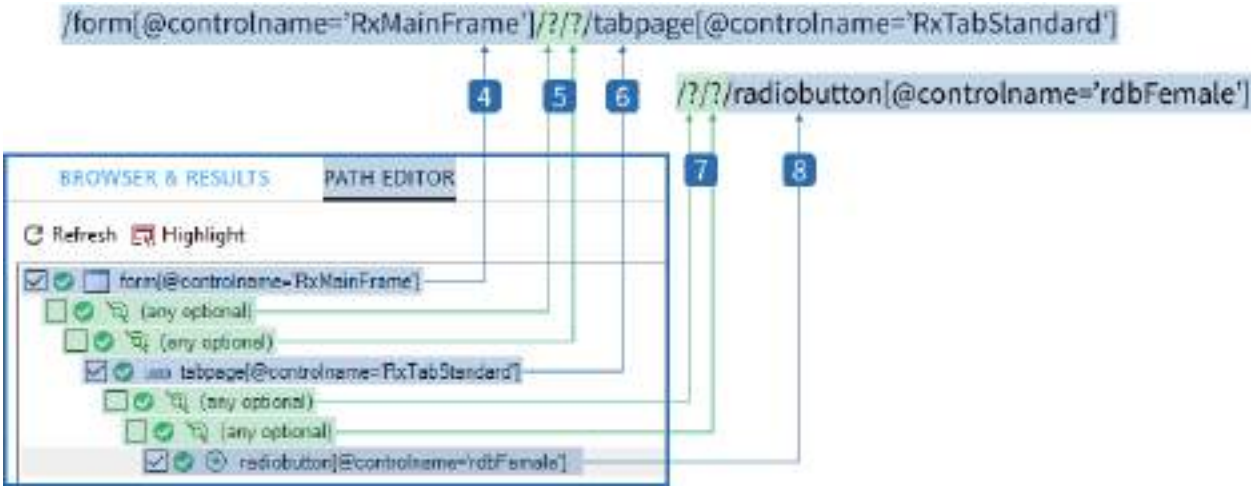
2 **Element tree browser view**

- Displays the hierarchical structure of the UI the radio button is embedded in.
- It contains five different UI elements.

3 **Path editor**

- Displays a visual representation of the nodes in the RanoreXPath expression.
- There are seven nodes in total, including the root and final node.

Let's take a closer look at the automatically generated RanoreXPath expression as it's displayed in the path editor:



4 **Root** `RxMainFrame`:
This node represents the program window of the application under test (i.e. the Ranorex Studio Demo Application) and is a fixed part of the RanoreXPath expression.

5 **Wildcards**:
To make the RanoreXPath expression more flexible, Ranorex Studio has inserted two wildcards `/?` (= any optional) between the root and the next fixed part. This means that between these two parts, either zero, one, or two nodes can be present and Ranorex Studio will still be able to find the final UI element.

6 **Fixed** node `RxTabStandard`:
Ranorex Studio has designated this UI element to be a fixed node that's necessary for identifying the radio button correctly.

7 **Wildcards:**
Ranorex Studio has inserted two more `/?` wildcards to further increase the flexibility of the RanoreXPath expression.

8 **Final node** `rdbFemale`:
The Female radio button is the final fixed part of the RanoreXPath expression; the actual UI element we want to identify.

Summary

- Wildcards increase the flexibility of a RanoreXPath. They allow Ranorex Studio to identify UI elements even if the UI structure they are embedded in changes. Wildcards can therefore increase robustness when applied correctly.
- In the example above, Ranorex Studio generated a RanoreXPath with three fixed nodes. These provide the required detail for accurately identifying the correct UI element (the **Female** radio button).
- Ranorex Studio also included four `/?` wildcards. These provide flexibility and ensure that changes to the UI structure between the fixed nodes won't break correct identification of the Female radio button.
- There are three different wildcard operators in total, explained below.

Wildcard operators

There are three different wildcard operators in Ranorex Studio:

`/*`	**any**	Any UI element in exactly one (1) tree level
`/?`	**any optional**	Any UI element in exactly zero (0) or one (1) tree level
`//`	**any descendants**	Any UI element in any number of tree levels

Reference

For more examples of using wildcards in RanoreXPaths, go to Ranorex Studio advanced > RanoreXPath > ⇢ RanoreXPath examples

RanoreXPath examples

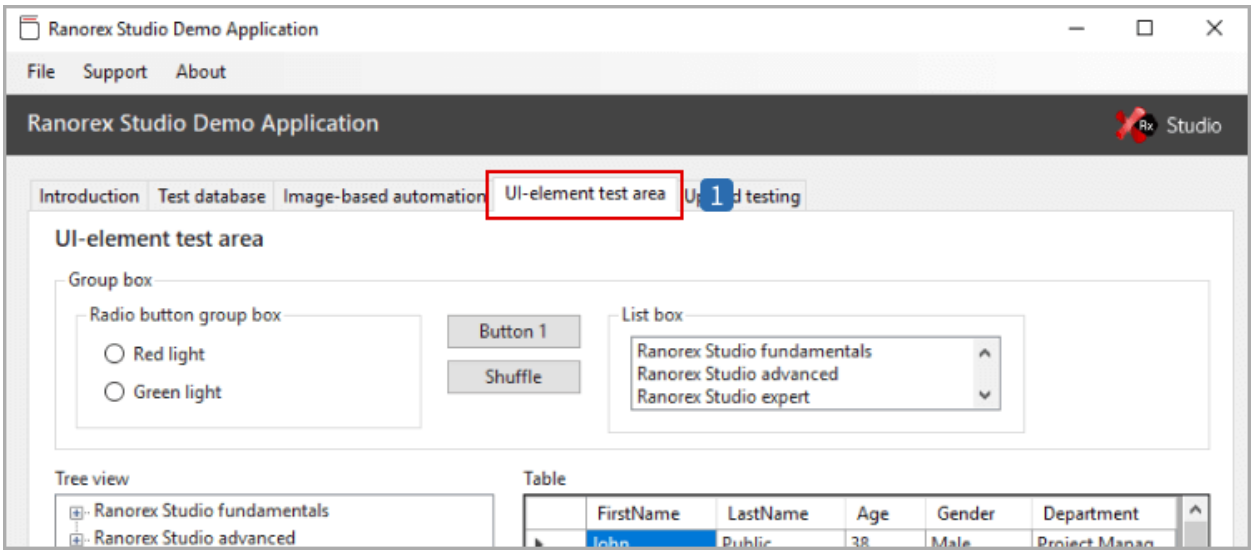
On this page, we've collected many detailed practical examples that show what a RanoreXPath expression for a specific scenario could look like.

Screencast
The screencast “Syntax examples” walks you through the information found in this chapter:

Watch the screencast now

Test example definition

All examples on this page are based on the Ranorex Studio Demo application. Most of them use the tab **UI-element test area** in this application.



1 **UI-element test area** tab in the Ranorex Studio demo Application

Download the Ranorex Studio Demo Application

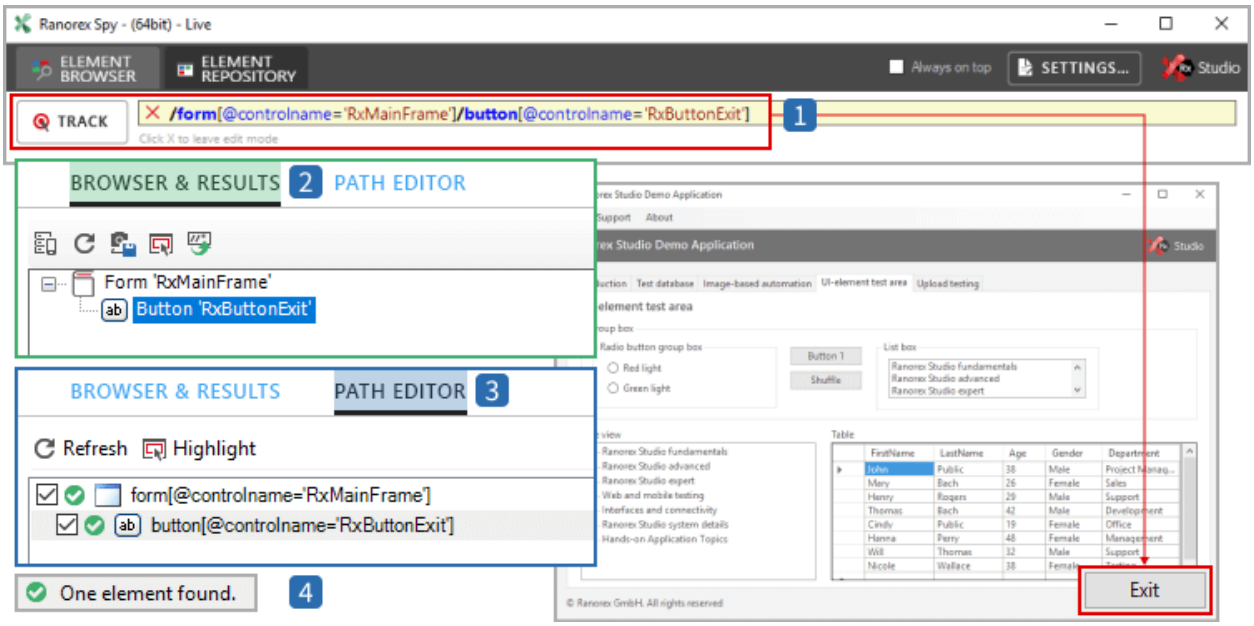
Follow the link below to download the file. Then unzip it and start the application.

Identify a simple button and generalize its RanoreXPath

Here we show you how to identify a simple button in a user interface and then generalize the corresponding RanoreXPath expression to match any button.

1 In Ranorex Spy, **track** the **Exit** button in the **UI-element test area** in the demo app.

After tracking, Spy shows the following:



1. The automatically generated RanoreXPath for the **Exit** button.
2. The tree browser shows that the **Exit** button is a child of the root UI element `RxMainFrame`.
3. The **Exit** button is identified by the attribute-value pair `@controlname='RxButtonExit'`.
4. The Spy status bar (bottom left corner) confirms **one** UI element found.

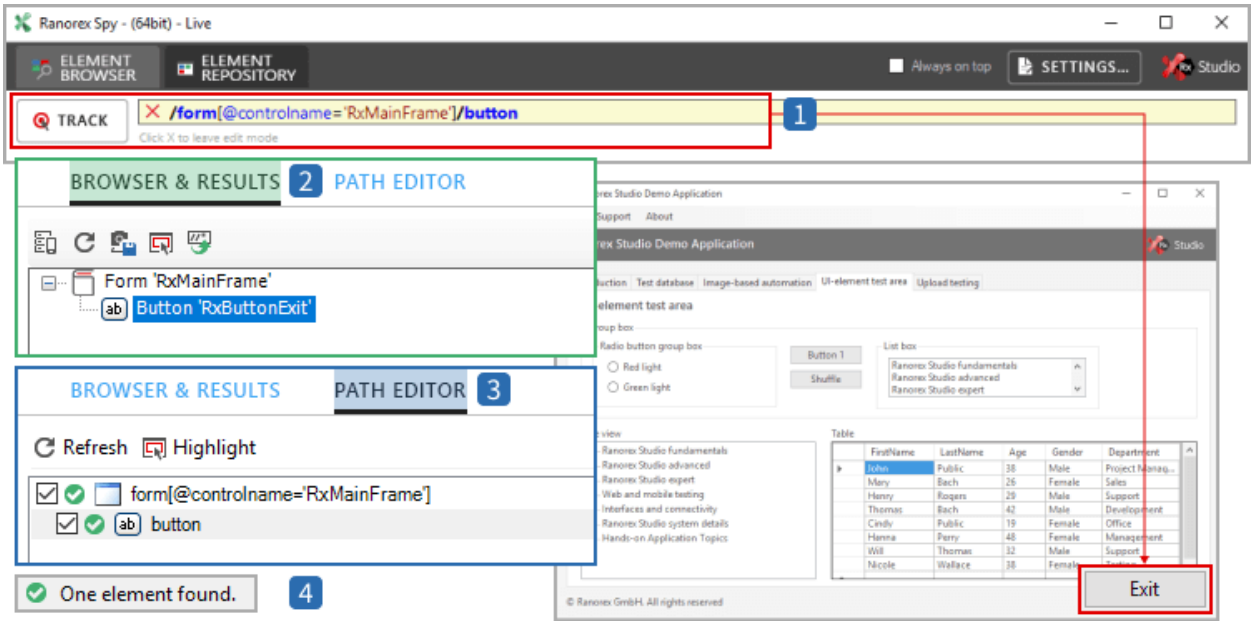
Generalize the RanoreXPath

Generalizing a RanoreXPath expression means making it less detailed. This can increase robustness.

In the current example, we want to generalize the RanoreXPath so that the **Exit** button is only identified by its `button` role. We do this by removing the button's attribute. Since the **Exit** button is the only button at this level in the element tree, it will still be identified correctly and its RanoreXPath more robust against UI changes.

1. In Spy, **click** into the RanoreXPath line, **delete** the attribute `[@controlname='RxButtonExit']`, and **press** Enter.

Spy will show the following:



1. The **generalized** RanoreXPath with only the `button` role.
2. The tree browser shows the **Exit** button as a child of the root element.
3. The path editor view represents the generalized RanoreXPath specification.
4. **One** UI element (i.e. the **Exit** button) has been found.

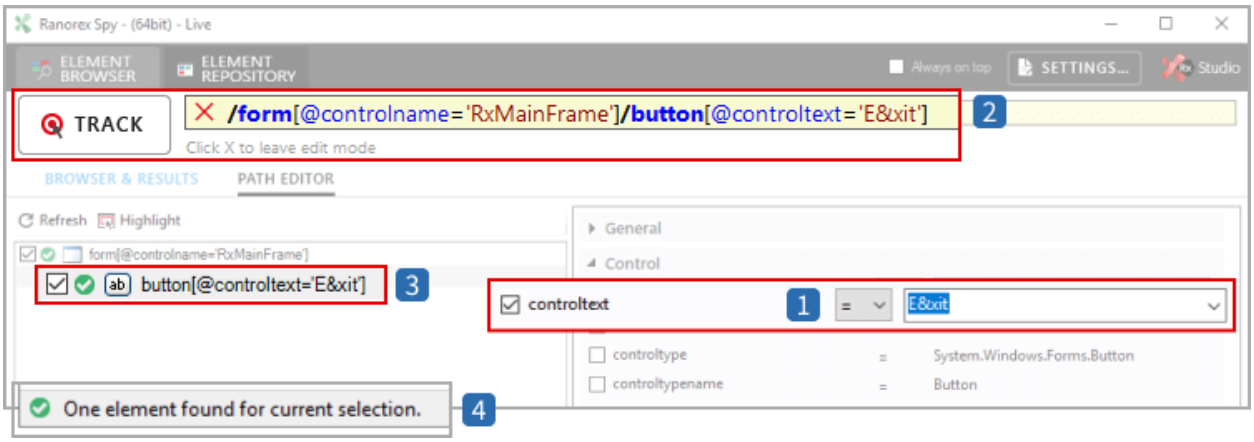
> **Note**
>
> If there is more than one button (or whichever type of UI element you generalized the RanoreXPath for) is present at the same level in the element tree, the RanoreXPath expression will identify all of them.

Change the identifying attribute

When tracking a UI element, Ranorex Studio automatically generates the RanoreXPath and therefore also determines identification attributes automatically. You can of course change the attribute by which a UI element is identified in a RanoreXPath.

1. **Track** the **Exit** button in the **UI-element test area** in the demo app.
2. **Click** into the RanoreXPath line, then **switch** to the Path Editor.
3. **Select** the node whose attribute you want to change, e.g. the button in this example.

4 On the left, **check**/**uncheck** the desired attributes, i.e. **deselect controlname** and **select controltext** instead for this example. The RanoreXPath is adjusted automatically.



1 The automatically determined controlname attribute was deselected and replaced with the controltext attribute.

2 The RanoreXPath reflects this change.

3 The path editor view also reflects this change.

4 The message in the status bar indicates that one element was found for this RanoreXPath.

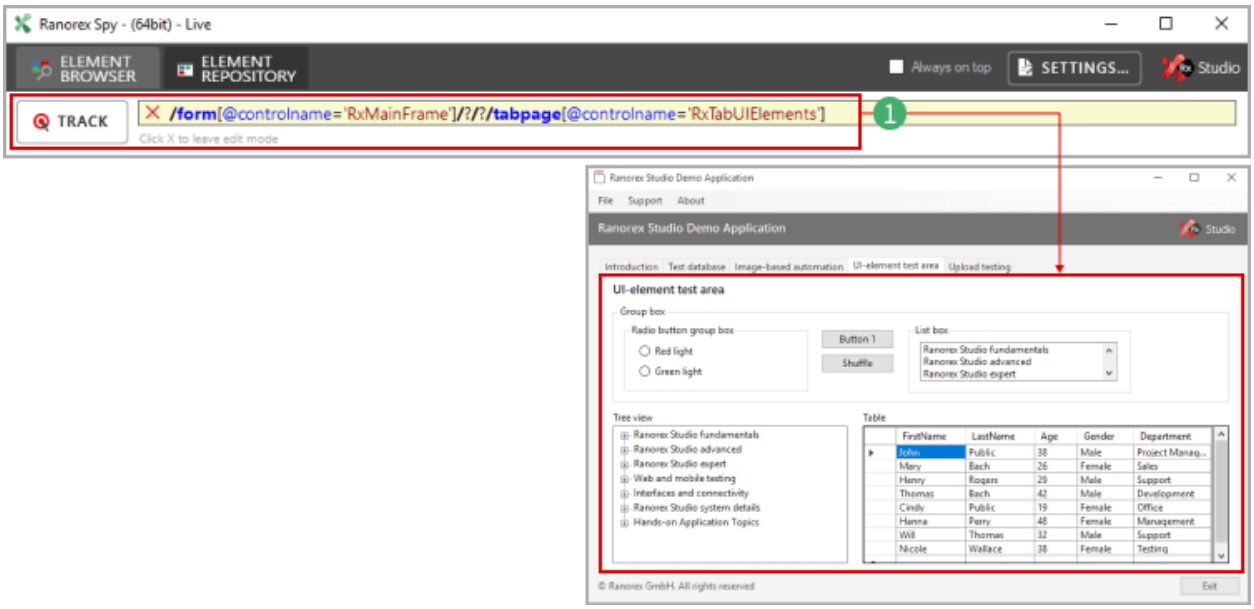
Wildcards

Wildcards can make a RanoreXPath more resistant to changes in the structure of the UI.

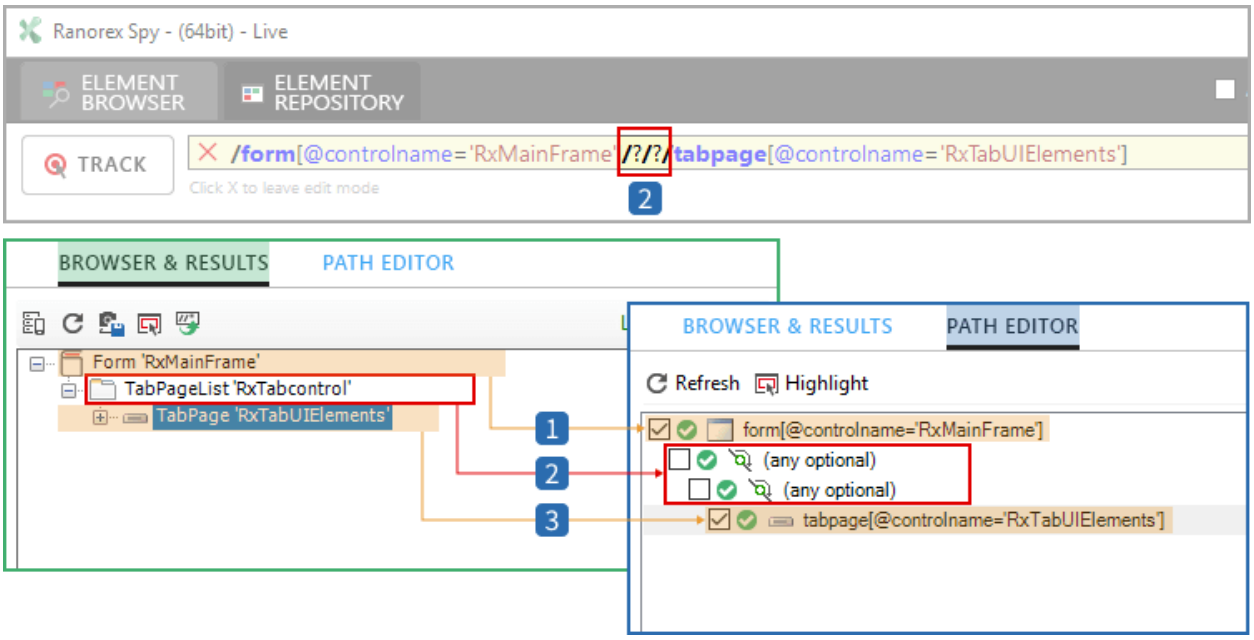
> **Note**
>
> The following wildcards exist:
>
> | | |
> |---|---|
> | `/*` | Any UI element, exactly one (1) tree level |
> | `/?` | Any UI element, exactly zero (0) or one (1) tree level |
> | `//` | Any UI element, any number of tree levels |

/? (any optional)

1 **Track** the entire UI-element test area in the demo application.

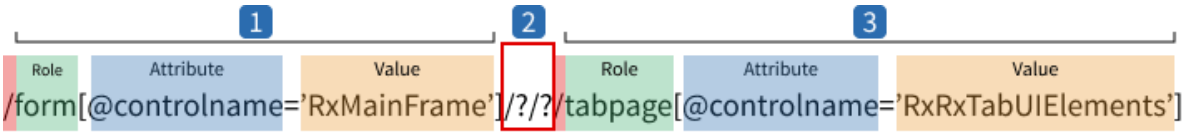


Ranorex Spy will show the following:



1. The **root** element is represented in the first fixed part of the RanoreXPath expression.
2. The middle UI element **TabPageList** is not represented. Instead, two `/?` wildcards replace it to make the path more robust. This is the wildcard part of the RanoreXPath.
3. The **final node** is the target UI element, i.e. the entire **UI-element test area** tab page. It is the second fixed part of the RanoreXPath.

Interpret the RanoreXPath

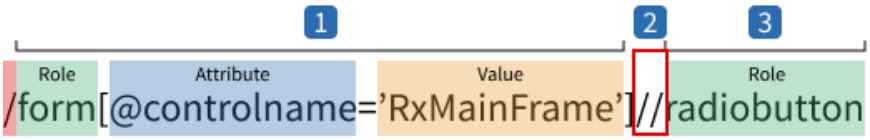


The 3 **target element** is a descendant of the 1 **root element** with a total of either 2 **zero, one, or two UI elements** between them on two different tree levels.

// (any descendants)

It may often be necessary to identify elements whose "depth" in the UI we don't know. The any descendants wildcard allows you to circumvent all the intermediate nodes between the root note and the final node.

1 **Enter** the following RanoreXPath expression in the tree browser in Ranorex Spy:

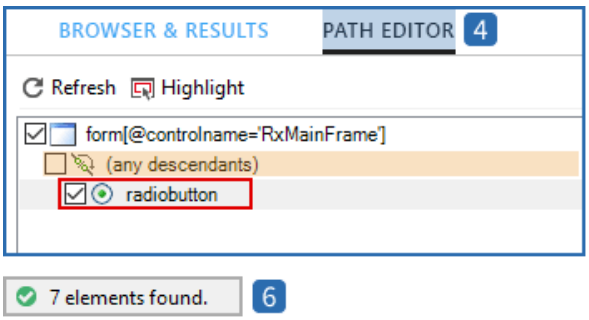


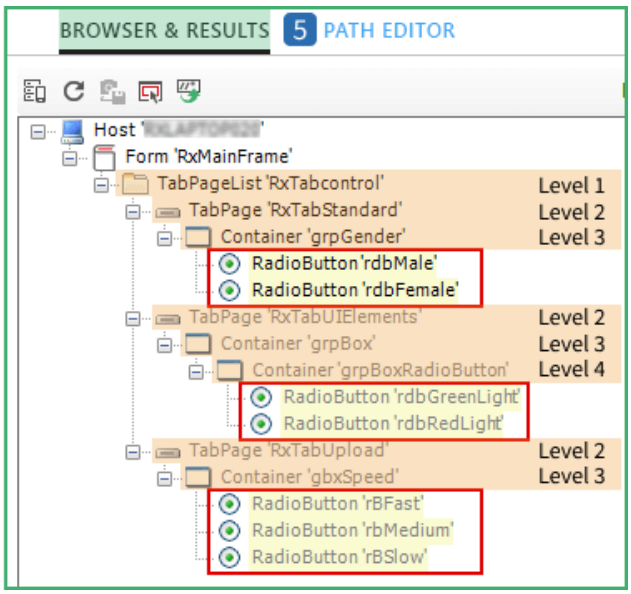
1 The first part of the RanoreXPath expression represents the root node, i.e. the demo app window. This is the first fixed node.

2 The second part is the any descendants wildcard. It is the wildcard node.

3 The third part is the final node that represents the target UI element. In this case, this is any UI element with the role `radiobutton`. The expression identifies all radio buttons that are descendants of the root element.

Spy shows the following results for this RanoreXPath:





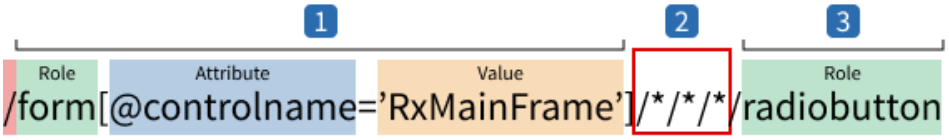
4 The RanoreXPath as represented in the **path editor**.

5 The **tree browser** displays seven radio buttons on two different levels (level 3 and level 4).

6 The **status message** confirms 7 elements found.

/* (any)

If you know at what "depth" in the UI an element is you want to identify and the intermediate nodes are irrelevant, then the any wildcard is a good choice for making your RanoreXPath more robust.

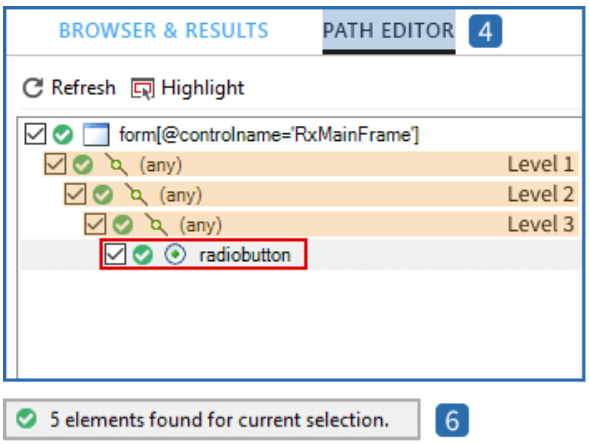


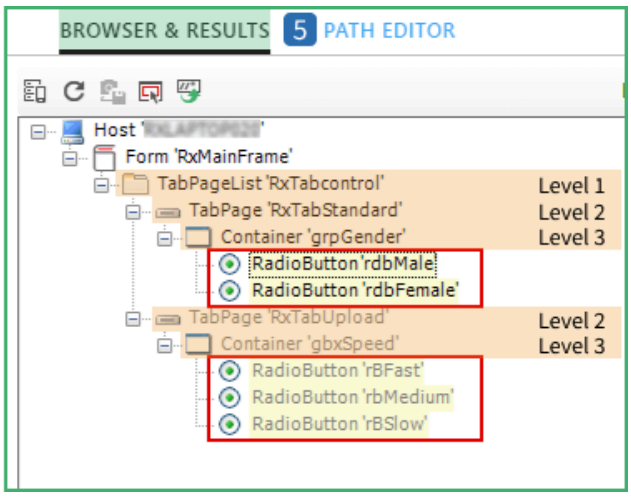
1 The first part of the RanoreXPath expression represents the root node, i.e. the demo app window. This is the first fixed node.

2 The second part consists of three any wildcards, representing any three UI elements on three different levels. This is the wildcard part of the expression.

3 The third part is the final node that represents the target UI element. In this case, this is any UI element with the role `radiobutton`. The expression identifies all radio buttons that are exactly three levels below the root element.

Spy shows the following results for this RanoreXPath:





4 The RanoreXPath as represented in the **path editor**. Each of the wildcard operators represents **one** level below the root element.

5 The **tree browser** shows five `radiobutton` elements exactly three levels below the root element. Any radio buttons at other levels are not identified.

6 The **status message** confirms five found UI elements.

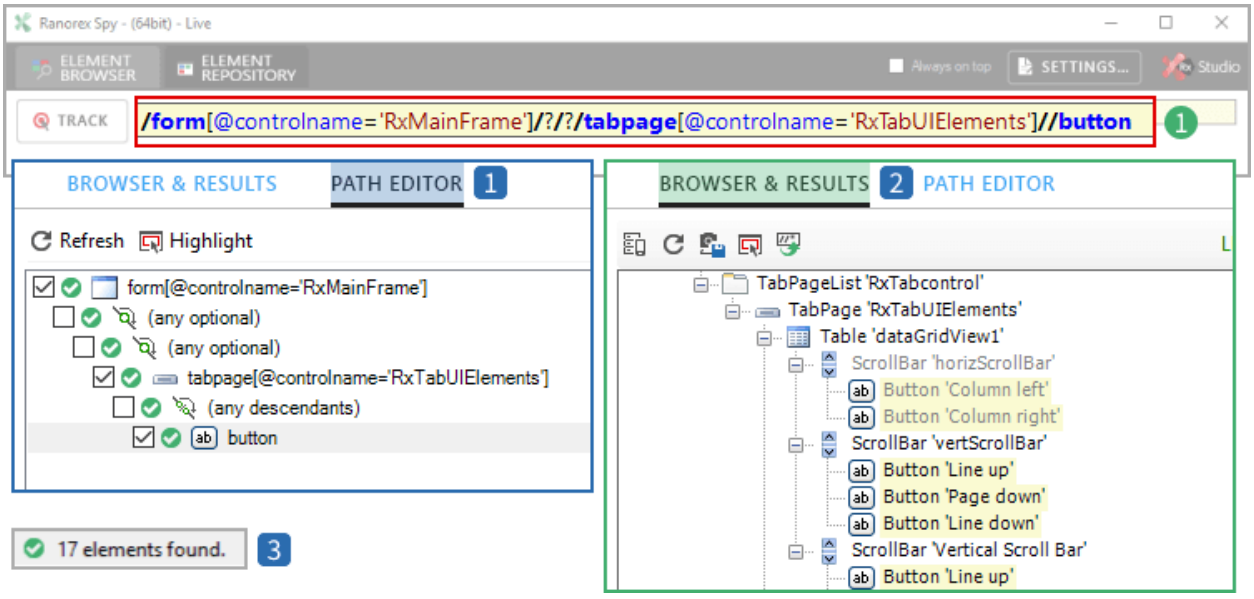
Make RanoreXPath expressions more detailed

RanoreXPath expressions can often be too general, resulting in too many or incorrect UI elements being identified. In these cases, you need to make the expression more detailed.

General RanoreXPath

For our example, let's first create a very general RanoreXPath. We'll simply identify all buttons in the UI-element test area tab of the Ranorex Studio Demo Application.

1. In Spy, **enter** the following RanoreXPath and **press** Enter.

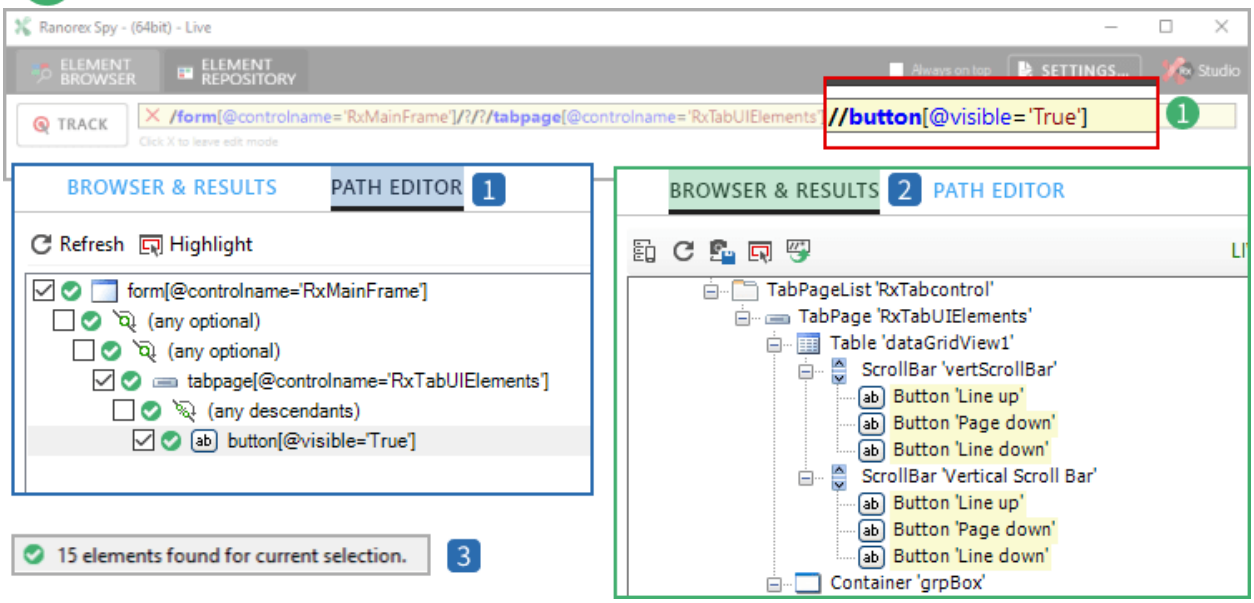


1. The RanoreXPath as represented in the path editor**.**
2. The **tree browser** shows 17 identified buttons.
3. The **status message** confirms 17 found UI elements.

Introduce more details

Seventeen buttons are clearly too many. We need to make the RanoreXPath more detailed. We do so by adding an attribute to the button. In this case, we use the `visible` attribute to identify only buttons that are not hidden.

1. **Change** the last part of the RanoreXPath as follows and **press** Enter.

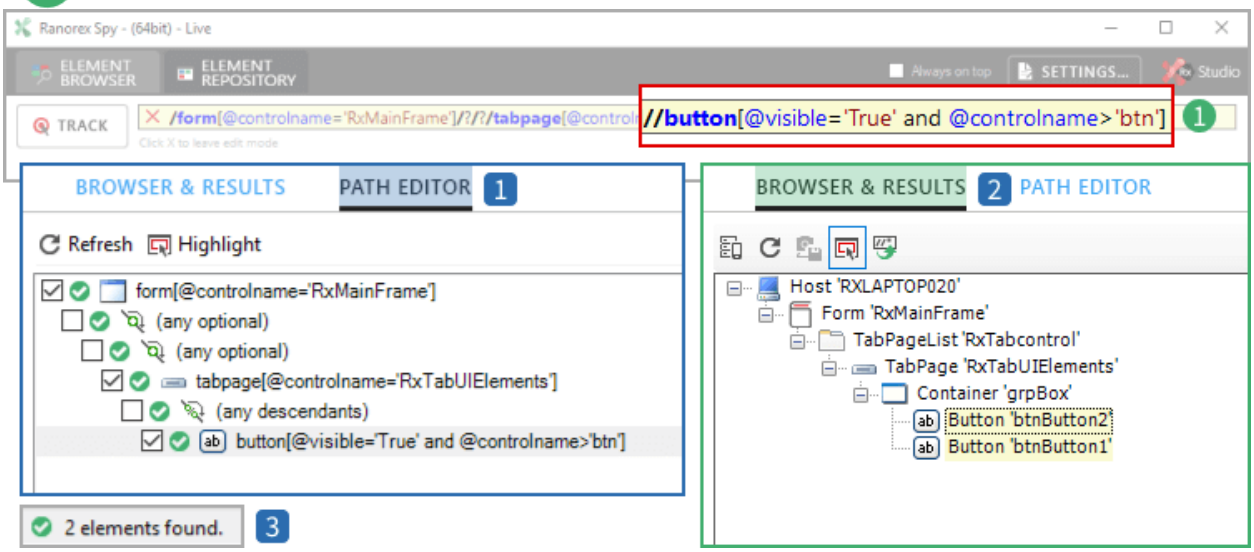


1. The new RanoreXPath as represented in the path editor.
2. The **tree browser** shows 15 buttons.
3. The **status message** confirms 15 UI elements found.

Even more details

We've only reduced the number of buttons by two, so we need to narrow down the RanoreXPath further. To do so, we can add another attribute and combine it with the previous one using the and operator.

1. **Change** the last part of the RanoreXPath as follows and **press Enter:**

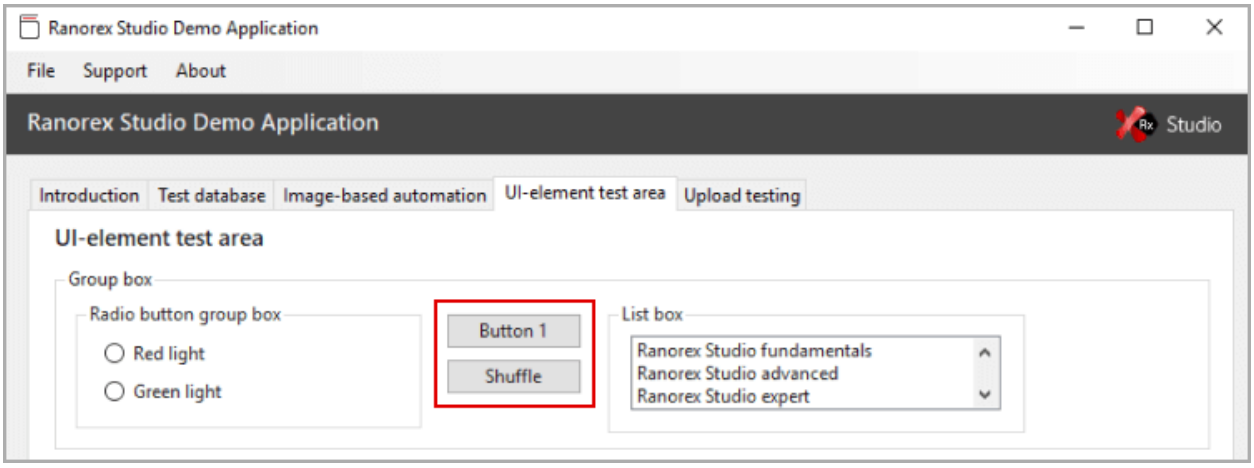


1. The **RanoreXPath** as represented in the path editor, now showing two attributes.
2. The **tree browser** now shows two UI elements.
3. The **status message** confirms two UI elements found.

> **Hint**
>
> The `and` operator is one of several available in the ⇢ RanoreXPath syntax.

Results

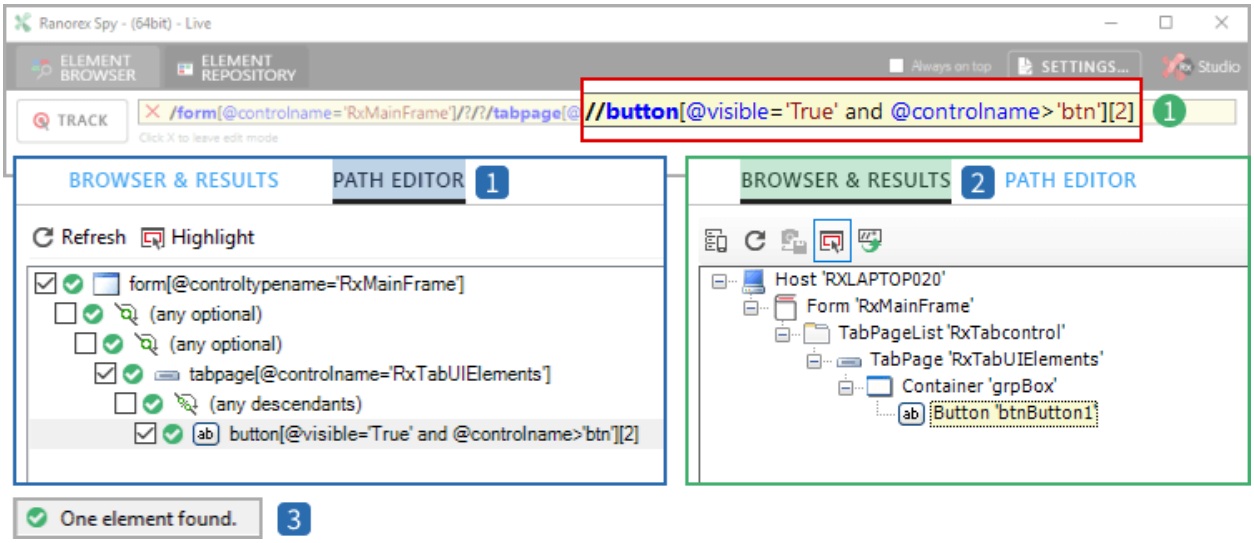
The more detailed RanoreXPath expression now only identifies the two buttons shown below.



Select UI elements based on tree position

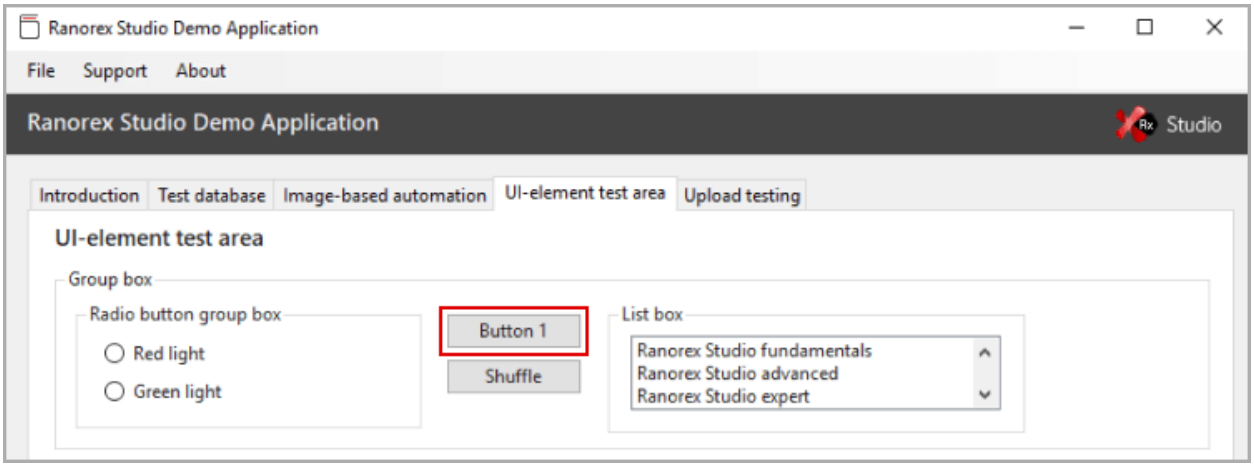
When a RanoreXPath identifies more than one element, you can select from among them by specifying their position in the element tree.

1. In Ranorex Spy, **enter** the following RanoreXPath and **press Enter**.



1. The RanoreXPath as represented in the **path editor.** The position operator `[2]` added after the predicate selects the **second** element from two elements identified by the preceding RanoreXPath.
2. The **tree browser** only shows the second UI element.
3. The **status message** confirms one found element.

In the UI of the demo app, this identifies the button **Button 1.**

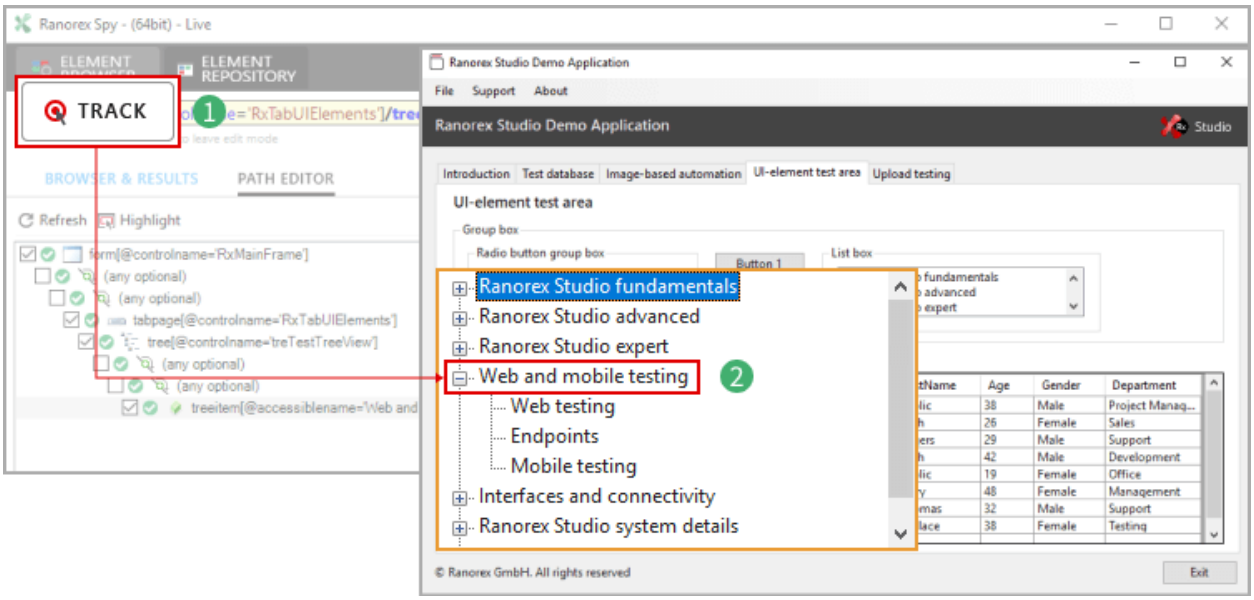


> **Note**
>
> **Only the position in the element tree is relevant.** In the UI itself, the position of the element may be different, as above with **Button 1**.

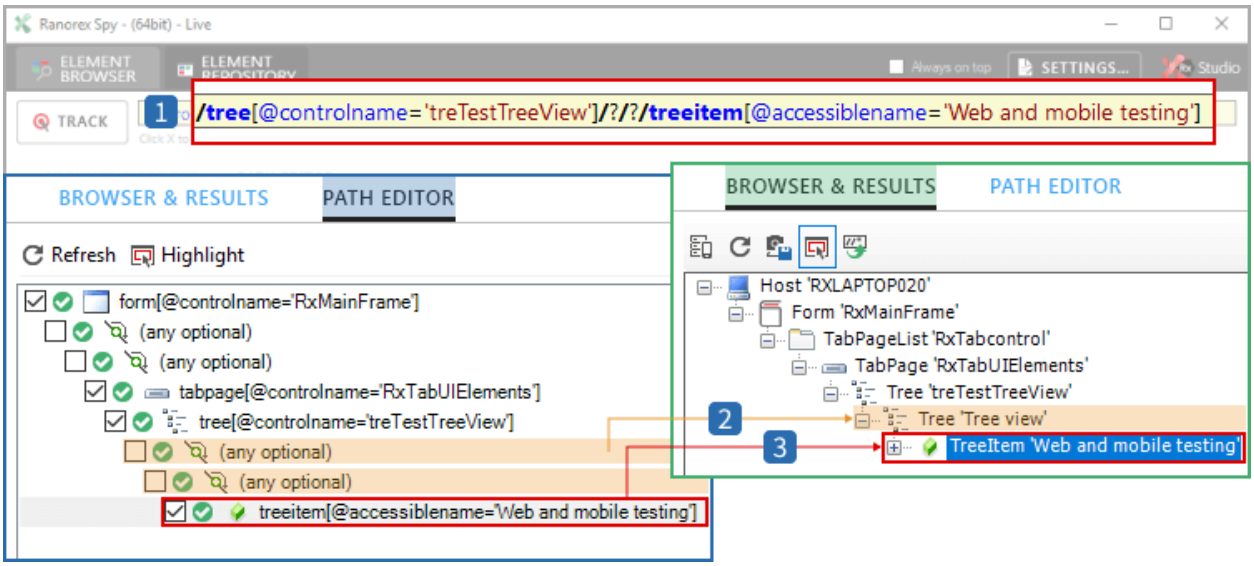
Identify tree view items

Tree views are a common UI element. Here we'll explain how Ranorex Studio identifies UI elements in them.

1. In Ranorex Spy, **click** the **Track** button.
2. In the UI-element test area in the demo app, **click** the **Web and mobile testing** tree item.



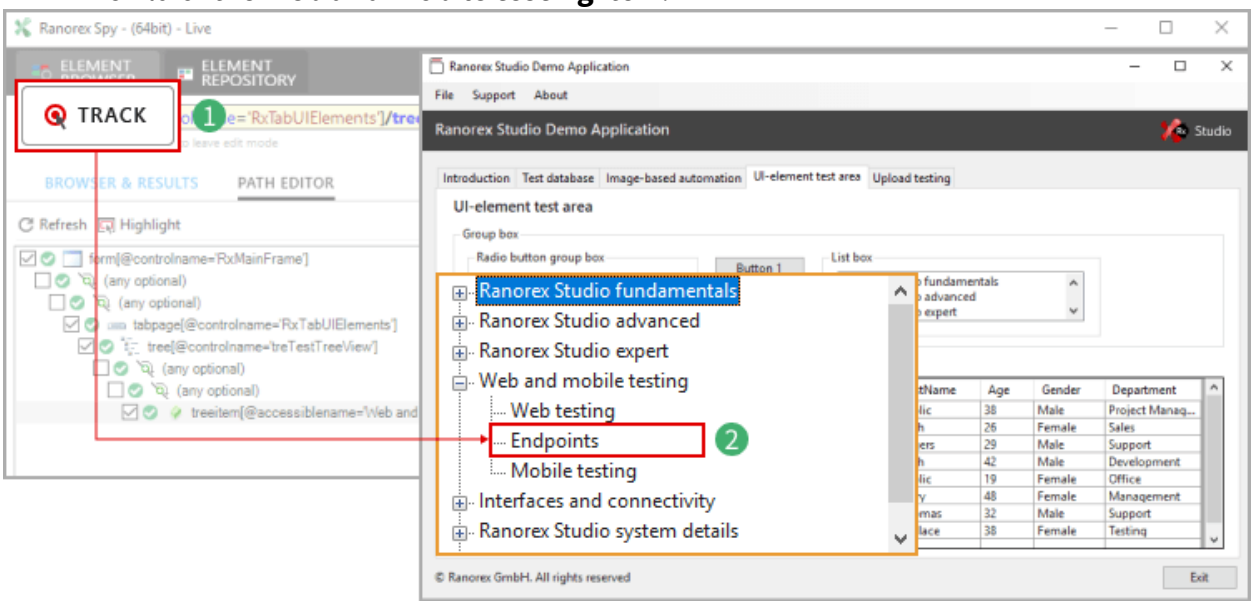
Ranorex Spy shows the following:



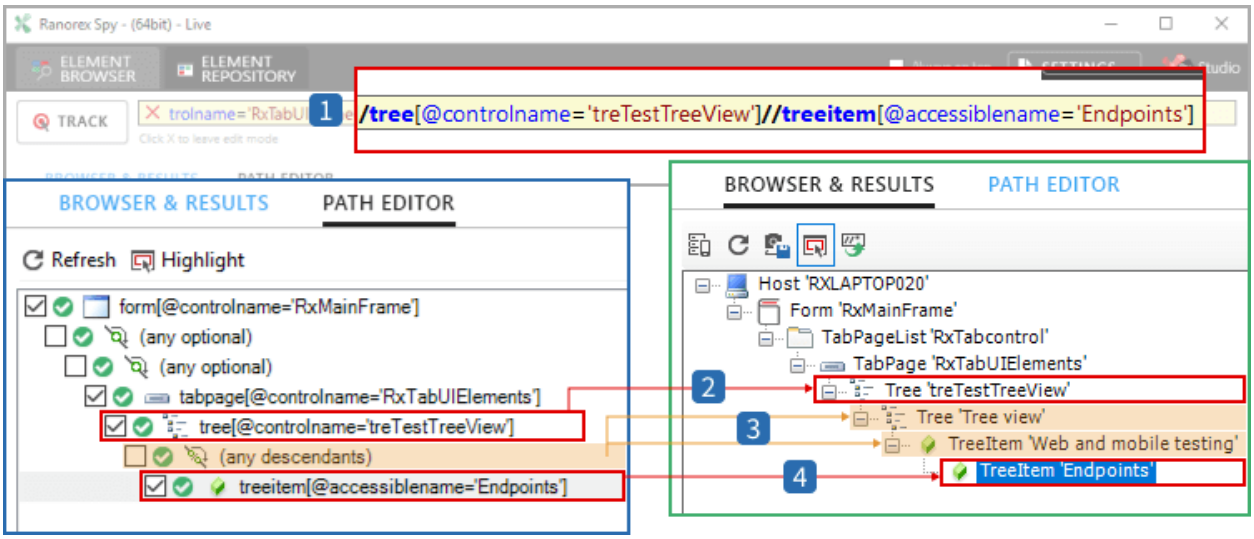
1. The **RanoreXPath for the tree item.**
2. Ranorex Studio automatically inserted **two 'any optional' wildcards** (`/?/?`) to increase robustness.
3. The element tree in Spy shows the tree item **Web and mobile testing** as a descendant of the tree view.

Subitems in tree views

1. In Ranorex Spy, **click** the **Track** button.
2. In the UI-element test area in the demo app, **click** the **Endpoints** tree item that is a child of the **Web and mobile testing** item.



Ranorex Spy shows the following:



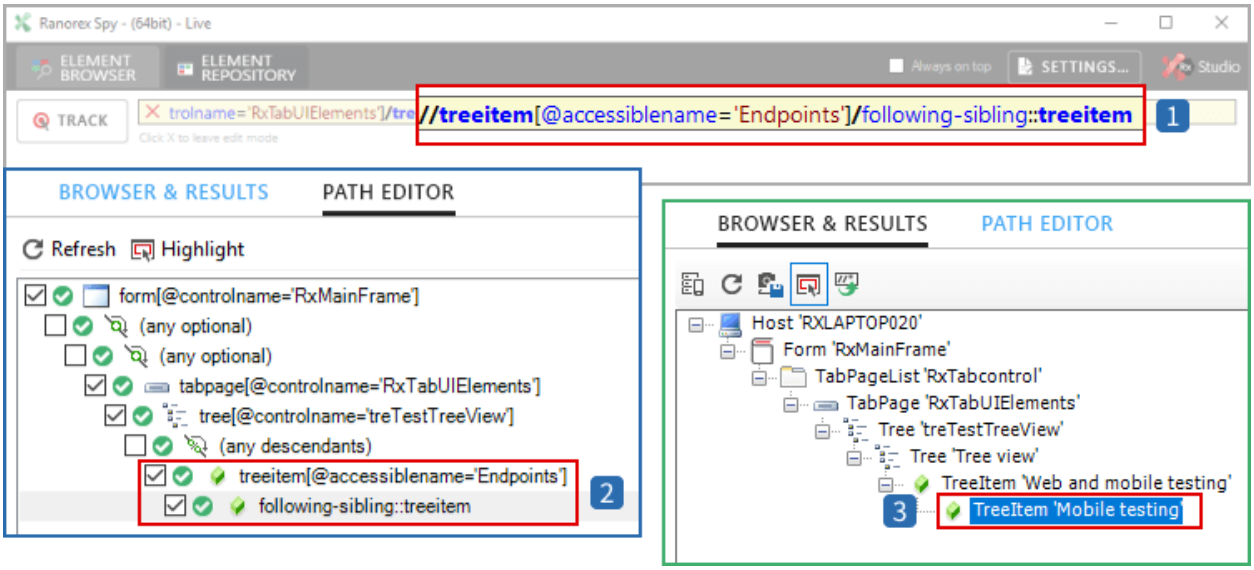
1. The RanoreXPath for the tree item. Note that the subtree item is identified in the same way as its parent, i.e. by its role and an attribute.
2. The ancestor tree view in its entirety serves as a fixed node.
3. Any tree items between the fixed node and the final node are bypassed by an any descendants `//` operator.
4. The subtree item Endpoints is identified regardless of how many ancestor tree items it has.

> **Note**
>
> Ranorex Studio treats all subtree items as regular tree items and identifies them with a predicate (attribute + value) in the same way. To ensure the tree structure does not interfere, it also inserts an any descendants operator. This way, it can find the tree item wherever in the tree it is.

Identify tree siblings with axes

When you want to identify the sibling(s) of a tree item, you can do so by using ⇢ axes. Axes allow you to specify nodes relative to a specific node. We'll illustrate this by using the `following-sibling` axis.

1. In Ranorex Spy, **enter** the following RanoreXPath and **press** Enter.

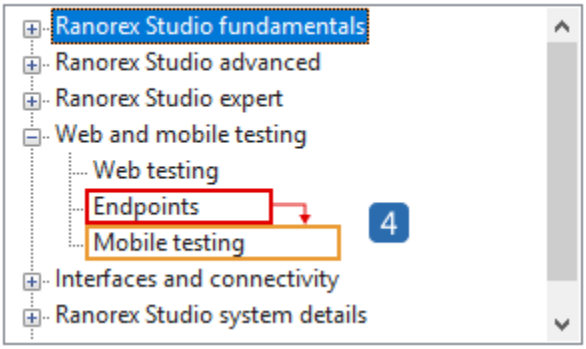


1 The RanoreXPath with the added axis `following-sibling::treeitem` after the part that identifies the Endpoints tree item. This axis means identifies all UI elements that have the role `treeitem` and are below the **Endpoints** tree item on the same level in Spy's element tree.

2 The RanoreXPath as represented in the **path editor**.

3 The **tree browser** in Spy displays the tree item **Mobile testing**, which is the only following sibling of the **Endpoint** item.

Tree view



4 The identification process illustrated in the UI of the demo app. The RanoreXPath first identifies the **Endpoints** item and then progresses to the **Mobile testing** item, which is the following sibling.

Note

`following-sibling` returns **all** following siblings if more than one exists.

Identify items in tables

Here we'll explain how identifying items in tables works with RanoreXPath.

Test definition

In this example we want to robustly identify the cell that contains the **age** of the person named **Thomas Bach** in the table of the UI-element test area in the demo app. This means that we need to identify both the cell that contains the name Thomas and, proceeding from it, the cell that contains the corresponding age.

Table

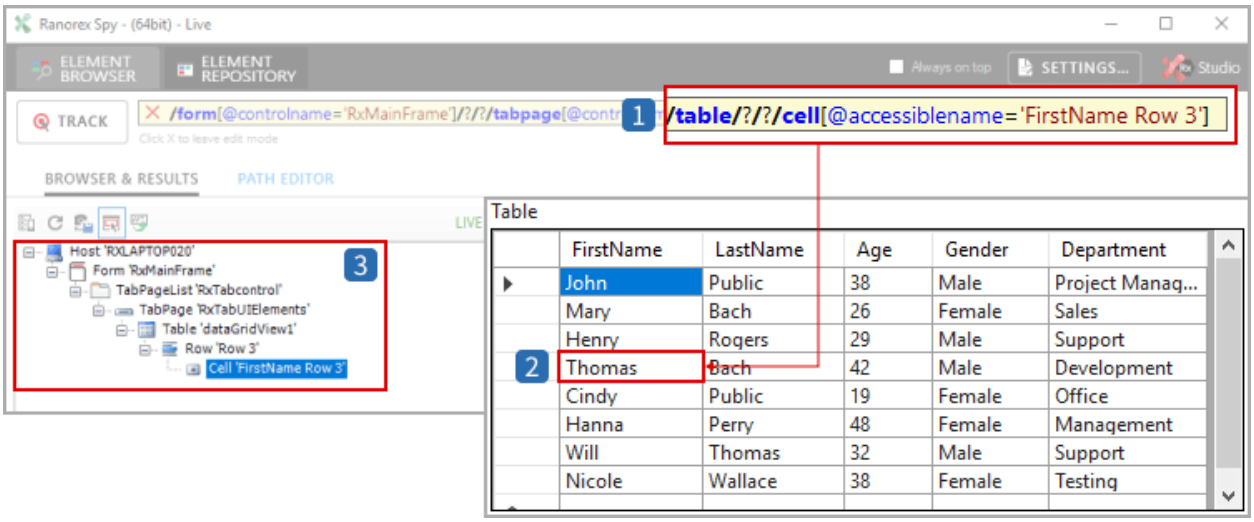
	FirstName	LastName	Age	Gender	Department
▶	John	Public	38	Male	Project Manag...
	Mary	Bach	26	Female	Sales
	Henry	Rogers	29	Male	Support
1	Thomas	Bach	42	Male	Development
	Cindy	Public	19	Female	Office
	Hanna	Perry	48	Female	Management
	Will	Thomas	32	Male	Support
	Nicole	Wallace	38	Female	Testing

1 The **row** with the details for Thomas Bach.

2 The **cell** that contains the age of Thomas Bach. We want to create a RanoreXPath that identifies it robustly.

Identify the name cell with an absolute RanoreXPath

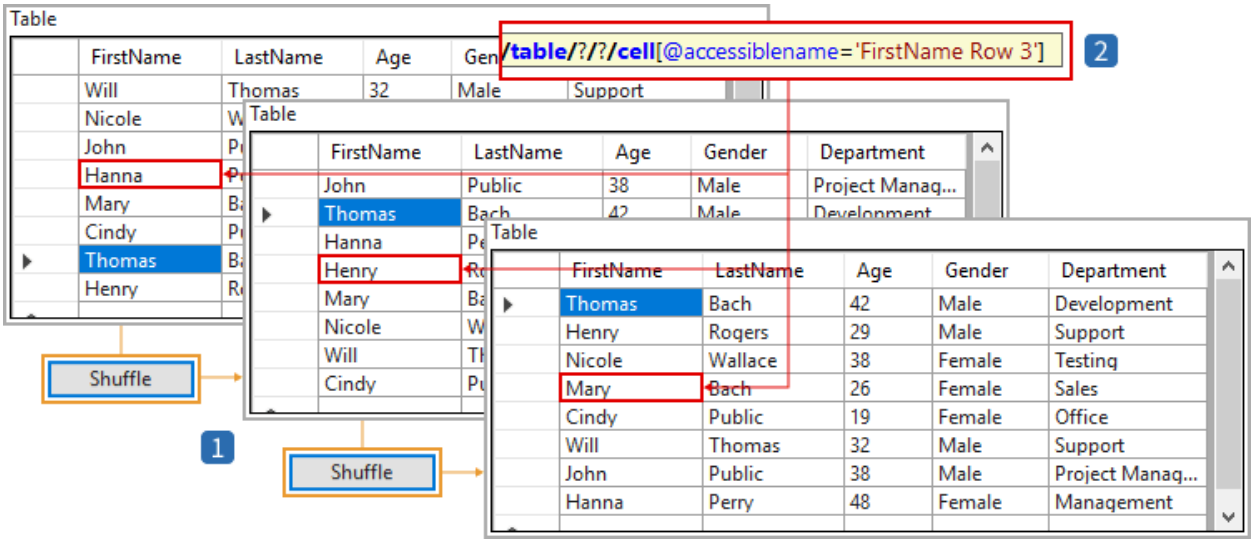
First we'll identify the cell that contains the name Thomas. A simple way is to just track the cell and generate an absolute RanoreXPath to it as in the image below.



1. **Absolute RanoreXPath** for the marked cell.
2. The RanoreXPath identifies the cell with predicate `@accessiblename='FirstName Row3'`, which reflects the cell's position in the table. This is the cell that currently has the value **Thomas**.
3. The **identified UI element** in the element tree in Spy.

Not robust against changes

The absolute RanoreXPath has its limitations here. When the table changes, the name in the identified cell may not be Thomas anymore, e.g. when you sort the table differently or shuffle it.

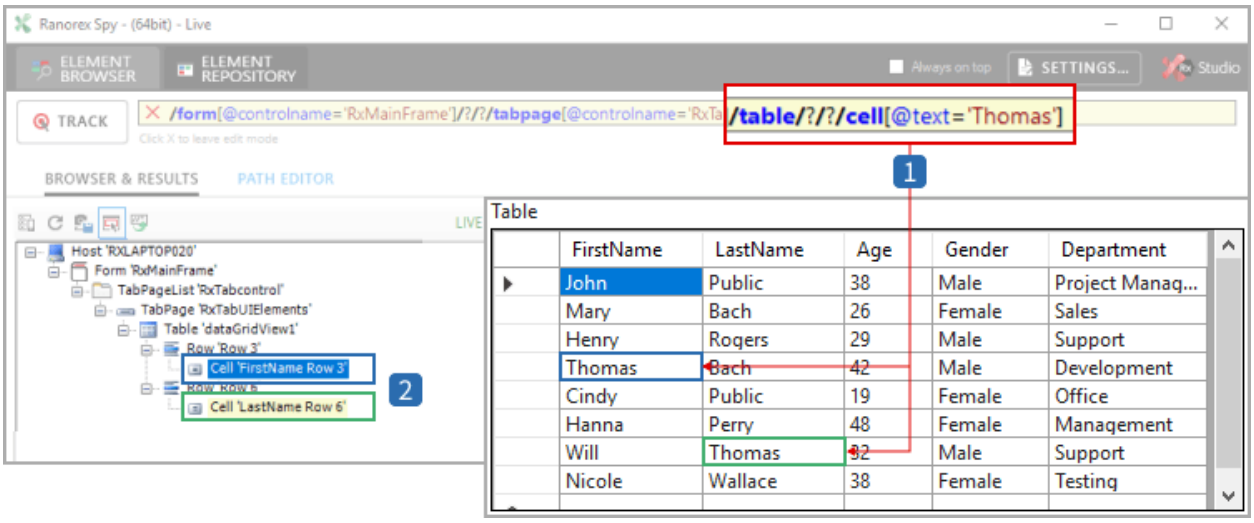


1. **Shuffling** the table cell arrangment shows that…
2. …the **absolute RanoreXPath** leads to wrong results, as it only cares about the cell position, not the name Thomas.

Specify cell content in the RanoreXPath

Instead of identifying the cell by its accessiblename (which is based on its position), we can use the attribute text. This attribute refers to the text in a cell.

This is still not a robust solution, however, as there are two cells with the text Thomas.

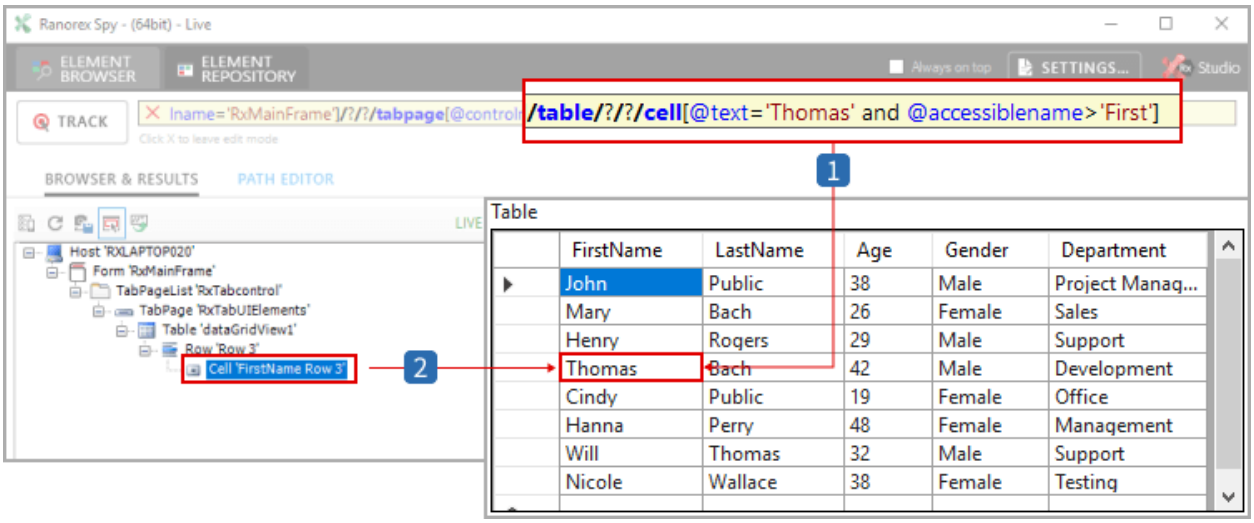


1 **Using the cell content** for identification makes the path robust against shuffling, but…

2 …also identifies all cells with **the specified content**.

Make the RanoreXPath more narrow

We need to make sure only the **Thomas** cell in the **FirstName** column is identified. We do so by extending the predicate by another attribute that relates to the accessiblename of the cell.



1 The narrowed cell with **two attributes** forming the RanoreXPath predicate.

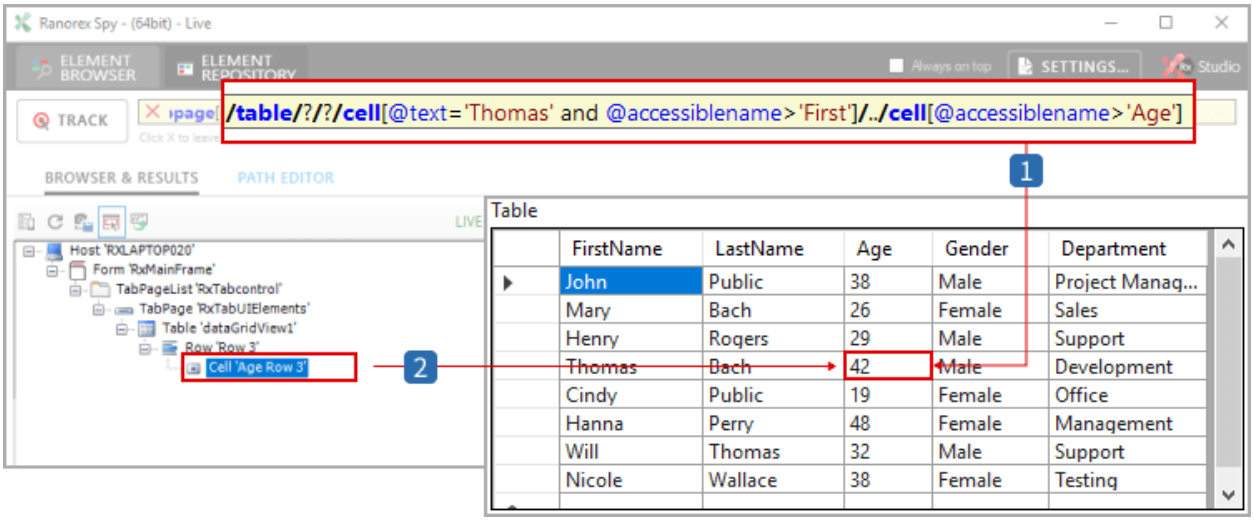
2 Only one cell is identified – the **Thomas** cell in the column **FirstName**.

> **Hint**
>
> Depending on the case, you may need to make the RanoreXPath even more detailed, e.g. by adding more attributes or with axes.

Going from absolute to relative

Now that we've robustly identified the cell containing the first name Thomas, we can now look at identifying the corresponding cell that holds Thomas's age.

We'll do so by making the path relative. This means that we'll use the `parent` axis (abbreviated: `..`) to navigate from an anchor point (the Thomas cell) back to the row and from there to the age cell in that row. This ensures the age cell is tied to the position of the Thomas cell and therefore always identified correctly.



1. The **relative RanoreXPath** with the parent axis and the age cell being identified by its accessiblename.
2. This identifies the cell with the **age of Thomas Bach (42)** correctly.

RanoreXPath Syntax

On this page, you'll find a list of the syntax expressions available. We'll first cover axes and then attribute expressions.

> **Reference**
>
> For more detailed examples of applied RanoreXPaths, please refer to ⇢ RanoreXPath examples.

Most of the examples on this page begin with `/form`, the role for top-level applications. Normally, you would further specify this role with an attribute, e.g. `[@controlname='RxMainFrame']` for the Ranorex Studio Demo Application, but on this page we omit this to keep the RanoreXPaths short.

Axes

An axis represents a relationship to the context (current) node, and is used to locate nodes relative to that node on the element tree. The context node is always the node before the axis.

All axes require further specification; they cannot stand alone. You can either specify a role (with or without a predicate) or use the any `/*` or any optional `/?` wildcards.

This does not apply to the abbreviations `//` (descendant-or-self) and `..` (parent). This is explained under the respective axis.

List of axes

Syntax
`child`

Description
Returns all children of the context node.

Example

```
/form/child::button
```

Syntax
`descendant`

Description
Returns all descendants of the context node.

Example

```
/form/descendant::button
```

Returns all descendants of `/form` that have the role `button`.

Syntax
`descendant-or-self | //`

Description
Returns the context node and all its descendants.
For the abbreviated syntax `//`, further specification is optional.

Example

```
/form/descendant-or-self::button | /form//button
```

Only returns the descendants of `/form` that have the role `button`. `/formhas` a different role and therefore isn't returned. If it had the role `button`, it would also be returned.

Syntax
`parent | ..`

Description
Returns the parent of the context node.
In its abbreviated syntax, it **must stand alone** and does not accept further specification.

Examples

```
/form/button[@caption='OK']/parent::container
```

```
/form/button[@caption='OK']/..
```

Example 1: Returns the parent of the specified button element if it has the role `container`.
Example 2: Returns the parent of the specified button element, regardless of its role.

Syntax
`ancestor`

Description
Returns all ancestors of the context node.

Example

```
/form//button/ancestor::container
```

Finds all buttons that are descendants of `/form` and then returns all ancestors of these buttons that have the role `container`.

Syntax
`ancestor-or-self`

Description
Returns the context node and all its ancestors.

Example

```
/form//button/ancestor-or-self::container
```

Finds all buttons that are descendants of `/form` and then returns all ancestors of these buttons that have the role `container`. If the context node also had the role `container`, it would be returned as well.

Syntax
`preceding-sibling`

Description
Returns all siblings before the context node.

Example

```
/form/button[@caption='OK']/preceding-sibling::button
```

Returns all UI elements that have the role `button`, are siblings of the specified button, and come before it.

Syntax
`following-sibling`

Description
Returns all siblings after the context node.

Example

`/form/button[@caption='OK']/following-sibling::button`

Returns all UI elements that have the role `button`, are siblings of the specified button, and come after it.

Attributes

Reference

You can also use regular expressions in RanoreXPaths. This is explained separately in Ranorex Studio Expert > ⇢ Regular expressions in Ranorex Studio.

Syntax
`/form`

Description
Top-level applications usually have the role `form`. This syntax therefore identifies top-level applications.

Syntax
`/form[@title='Calculator']`

Description
Identifies top-level applications that have the title Calculator.

Syntax
`/form[@title='Calculator' and @instance='2']`

Description
Identifies the top-level application that has the title Calculator AND the instance attribute 2.

Syntax
`/form[@title='Calculator' or @class='SciCalc']`

Description
Identifies top-level applications that have the title Calculator OR the class SciCalc.

Syntax
`/button`

Description
Identifies UI elements that have the role `button`.

Syntax
`/button[2]`

Description
Identifies the second button (refers to the position in the element tree).

Syntax
`/button[-2]`

Description
Identifies the second-to-last button (refers to the position in the element tree).

Syntax
`/button[@text='Exit']`

Description
Identifies buttons with the text Exit.

Syntax
`/button[@text!='Exit']`

Description
Identifies buttons with the text Exit.

Syntax
`/button[@text>'Warning']`

Description
Identifies buttons whose text begins with Warning.

Syntax
`/button[@text<'Warning']`

Description
Identifies buttons whose text ends in Warning.

Syntax
`/*[@text='Warning']`

Description
Identifies any element with the text Warning.

Syntax
`/button[?'Warning']`

Description
Identifies buttons that have any attribute that contains the string Warning.

Syntax
`/button[@text!=null()]`

Description
Identifies buttons whose text attribute is not empty.

Syntax
`/progressbar[@value>='13.5']`

Description
Identifies progress bars with a value greater than 13.5. Other possible operators: `>`, `<`, `>=`, `<=`.

Syntax
`/button[@text=$var]`

Description
Identifies buttons whose text has the value of the variable `$var`.

Syntax
`/button[$var]`

Description
Identifies the first, second, third, etc. button, depending on the numeric value of `$var`.

Syntax
`/dom//input[#'Search']`

Description
Identifies an input element on a website (`/dom` for document object model) with the unique identifier Search.

Functions

Syntax
`/table/row/cell[first()='True']`

Description
Identifies the first cell of a row.

Syntax
`/table/row/cell[last()='True']`

Description
Identifies the last cell of a row.

Syntax
`/table/row/cell[index()='2']`

Description
Identifies the second cell (refers to the position in the element tree). The `index()` function starts at 1.

Syntax
`/table/row/cell[pos()='2']`

Description
Identifies the third cell (refers to the position in the element tree). The `pos()` function starts at 0.

Syntax

```
/form[x()>'100' and y()='100']
```

Description

Identifies top-level applications with screen coordinates greater than 100 pixels.

Syntax

```
/form/button[cx()>'10' and cy()>'10']
```

Description

Identifies buttons with client coordinates (= an element's coordinates relative to its parent) greater than 10 pixels.

Syntax

```
/form[width()>'100' and height()>'100']
```

Description

Identifies top-level applications with a width and height greater than 100 pixels.

Syntax

```
/dom/body/form/button[text()='Clear']
```

Description

Identifies buttons on a website with the innertext Clear. For web testing, the function `text()` returns the value of the attribute innertext.

Syntax

```
/dom/body/form/[@id=null()]
```

Description

Identifies website forms whose ID attribute is not zero.

Image-based testing

In GUI test automation, there are different approaches to identifying UI elements. Ranorex Studio's main approach is based on object-oriented testing. This means it is able to directly ⇢ identify and interact with individual UI elements based on the technology they were implemented with.

However, in some cases, this approach may not work that well. For this reason, Ranorex Studio also supports image-based testing. This approach identifies and interacts with UI elements based on the recognition of pixels in an image.

In this chapter, you'll find out when and how to use image-based testing in Ranorex Studio and how to overcome the challenges of this testing approach.

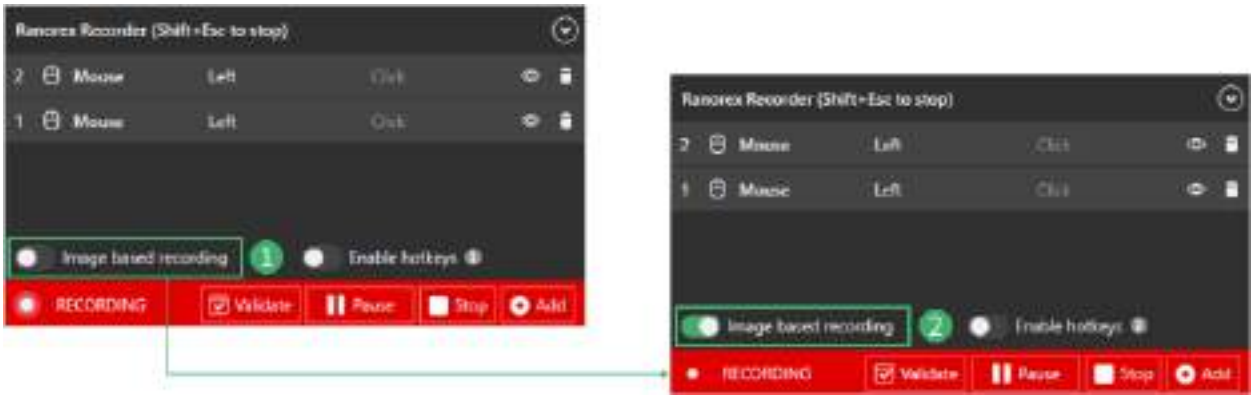
Enable image-based recording

The best way to use image-based testing in Ranorex Studio is through the Recorder. There, you can enable and disable image-based testing while you're recording. This way, you can combine object-oriented and image-based approaches in a single recording if required.

> **Attention**
>
> If you want to use image-based testing, ensure the setting **Asynchronous dispatching of mouse and keyboard events** is enabled. In most cases, disabling it will cause errors in image-based testing. Enable the setting in Ranorex Studio under **Settings > Advanced**.

To enable image-based recording:

1. While recording, **click** the **Image-based recording** switch.
2. Image-based recording is now active.



Alternatively, you can also use hotkeys:

3. While recording, **enable** hotkeys.
4. **Press** I to enable image-based recording.

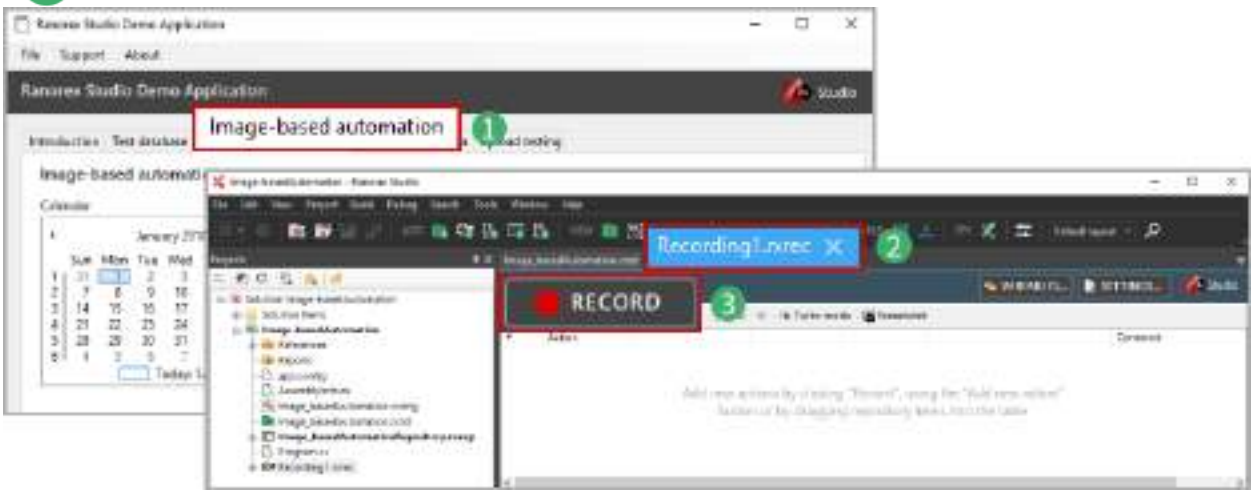
Why image-based testing?

Image-based testing is useful when the object-oriented approach can't produce satisfactory results. Here we'll show you an example where this is the case.

Example preparation

We'll use the Ranorex Studio Demo Application for our example and we'll record our actions in the Recorder.

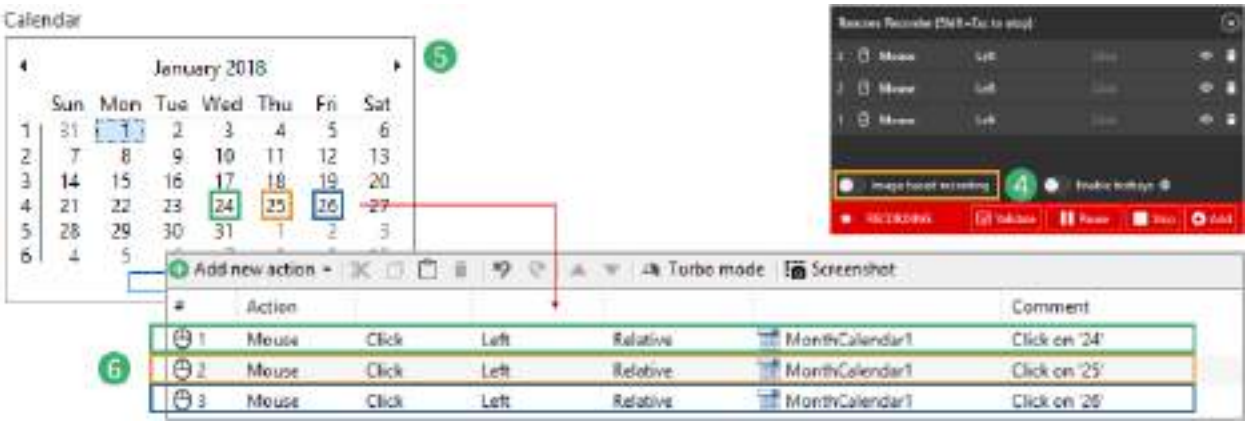
1. **Start** the demo application and **click** the **Image-based automation** tab.
2. In Ranorex Studio, **create** a new desktop solution with the solution wizard and **open Recording1** in this solution.
3. In the recording module view of **Recording1**, **click RECORD**.



Object-oriented recording

We'll now record mouse clicks on three consecutive dates in the calendar view of the demo application using the default object-oriented approach.

4. **Ensure** image-based recording is disabled.
5. **Click** the three dates **24**, **25**, and **26** in the calendar and **stop** the recording.
6. The action table displays the three corresponding actions.



Run the test

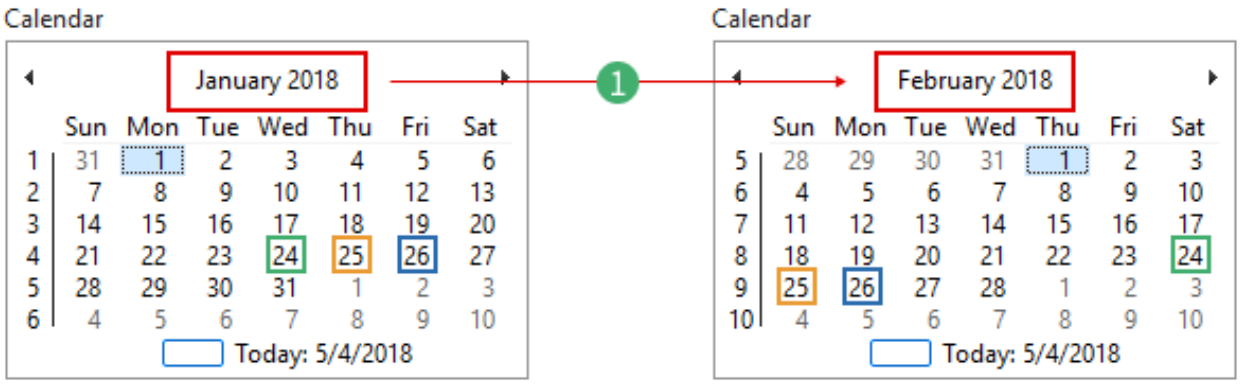
When you run the test, you'll see that

- the recorded calendar dates are identified correctly.
- each click action in the report corresponds to the correct date in the calendar.

Change of test conditions

So far, the object-oriented approach has worked flawlessly. Now, assume that you run the test a couple of months from its creation. This means that the calendar view will have changed to a different month. Let's see how our test performs in this case.

1. In the calendar in the demo application, **switch** to any month where the recorded dates 24, 25, and 26 are at a different position in the calendar.



2. **Run** the test again and **see** what happens.

- The test **runs without failure** and reports a success.
- However, **this is a false positive.** The test identifies the wrong dates (21, 22, and 23 in our case instead of 24, 25, and 26).

> **Hint**
>
> This is because sometimes, Ranorex Studio can't identify an individual UI element (e.g. dates within certain calendar views). Instead, it then uses absolute and relative positions rather than the actual object, which leads to the above issue.

Conclusion

On the next page, we'll show you how you can overcome this challenge with image-based testing.

Download the sample solution

You can download the completed solution with all the above steps carried out below:

Sample Image Based

Install the sample solution:

1. **Unzip** to any folder on your computer.
2. **Start** Ranorex Studio and **open** the solution file `ImagebasedAutomation.rxsln`

> **Hint**
> The sample solution is available for Ranorex versions 8.0 or higher. You must agree to the automatic solution upgrade for versions 8.2 and higher.

Image-based testing basics

On the previous page, we showed you a case where object-oriented testing fails to identify the correct UI elements. On this page, we'll show you the basics of image-based testing to solve this issue.

The image-based approach

As on the previous page, we want to test the following:

In the calendar view of the Ranorex Studio Demo Application, we want to perform mouse clicks on three different dates (24, 25, and 26) of a month. After changing the month, the mouse clicks should still be performed on the same dates.

1. **Start** the demo application and **click** the **Image-based automation** tab.
2. In Ranorex Studio, **create** a new desktop solution with the solution wizard and **open Recording1** in this solution.
3. In the recording module view of **Recording1**, **click RECORD**.

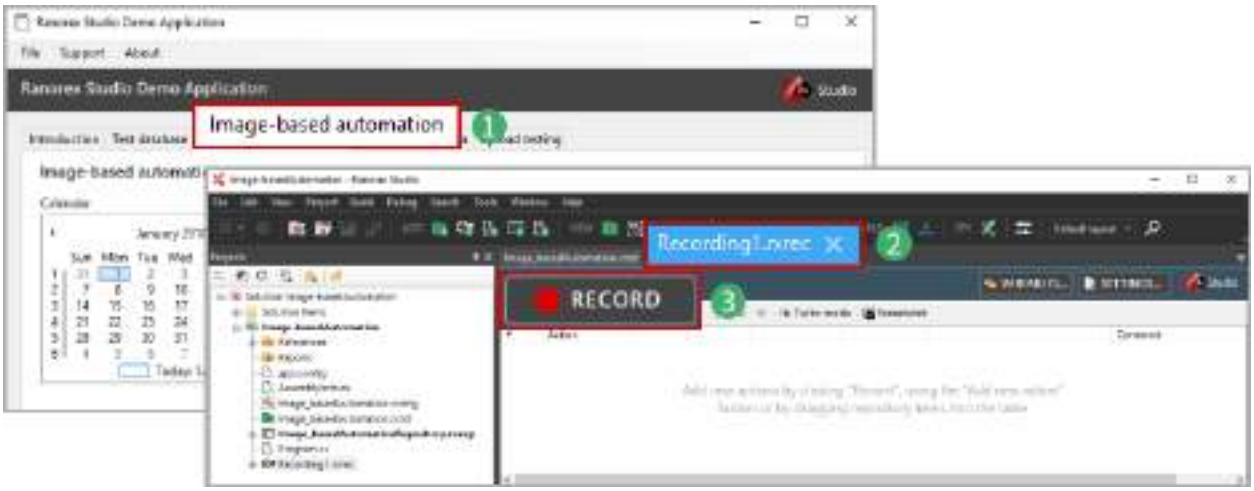
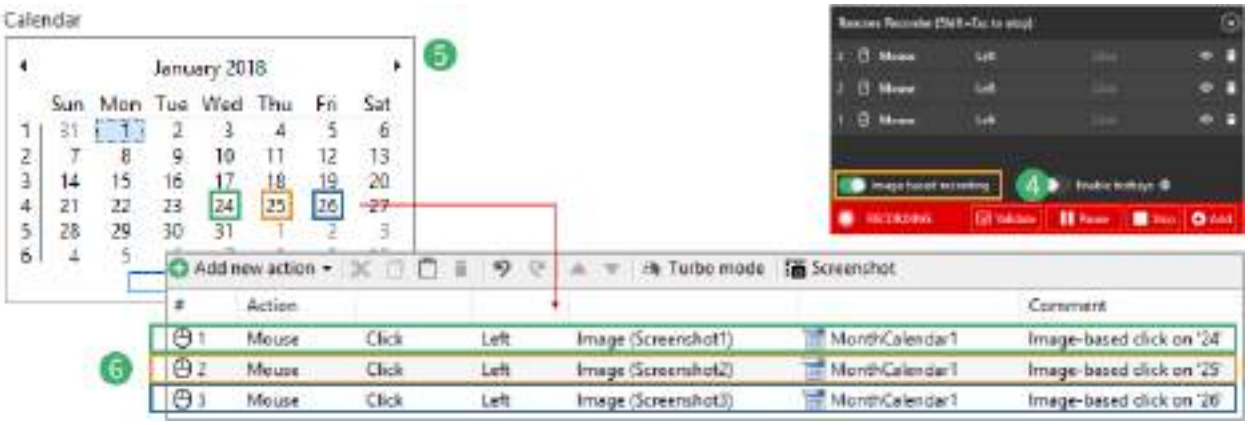


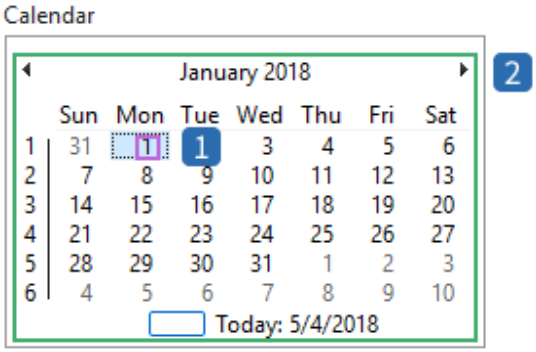
Image-based recording

We'll now record mouse clicks on three consecutive dates in the calendar view of the demo application using the **image-based approach.**

4. **Ensure** image-based recording is disabled.
5. **Click** the three dates **24**, **25**, and **26** in the calendar and **stop** the recording.
6. The action table displays the three corresponding actions.



> **Note**
>
> When image-based recording is enabled, Ranorex Recorder frames UI elements in green instead of purple (object-oriented recording), as shown in the image below.



1. **Object-oriented** recording with purple frame
2. **Image-based** recording with green frame

Run the test

When you run the test, you'll see that

- the recorded calendar dates are identified correctly.

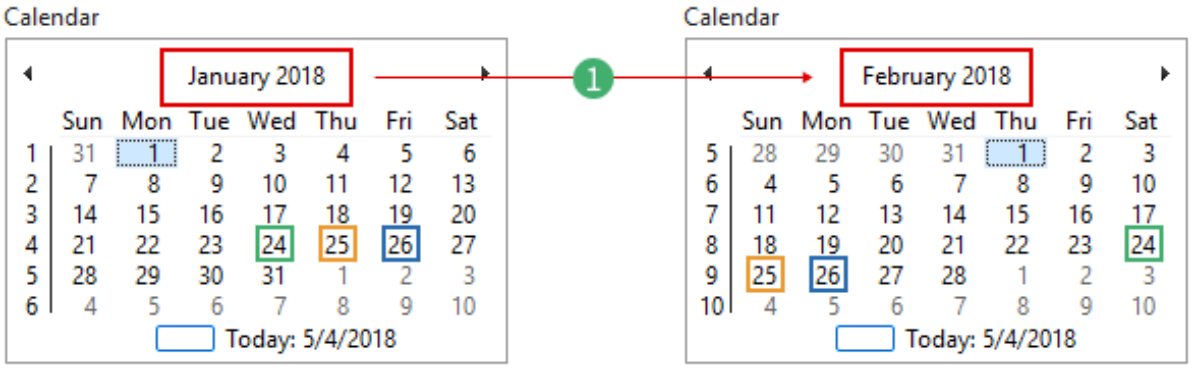
- each click action in the report corresponds to the correct date in the calendar.

In other words, at this point the test results are the same as with object-oriented testing.

Change of test conditions

Now assume that you run the test a couple of months from its creation. This means that the calendar view will have changed to a different month. Let's see how our image-based test performs in this case.

1. In the calendar in the demo application, **switch** to any month where the recorded dates 24, 25, and 26 are at a different position in the calendar.



2. **Run** the test again and **see** what happens.

- The test **runs without failure** and reports a success.
- Ranorex Studio has identified the correct dates despite their changed positions.
- Image-based testing has solved the issue created by object-oriented testing.

Advanced image-based testing

As we've seen on the previous page, image-based testing is sometimes the solution to an issue. However, as with any other testing approach, you need to apply it correctly or it will fail.

Proper image-based testing can get challenging when the recorded and the actual image during test execution don't match. This can be caused by simple things like a UI element being highlighted or icons changing during test execution. False positives, can also be an issue, when instead of the intended UI element, a similar-looking one is identified.

On this page, we'll show you how to use advanced image-based settings to solve these two issues. At the end of the page, you'll

Issue: Image not found

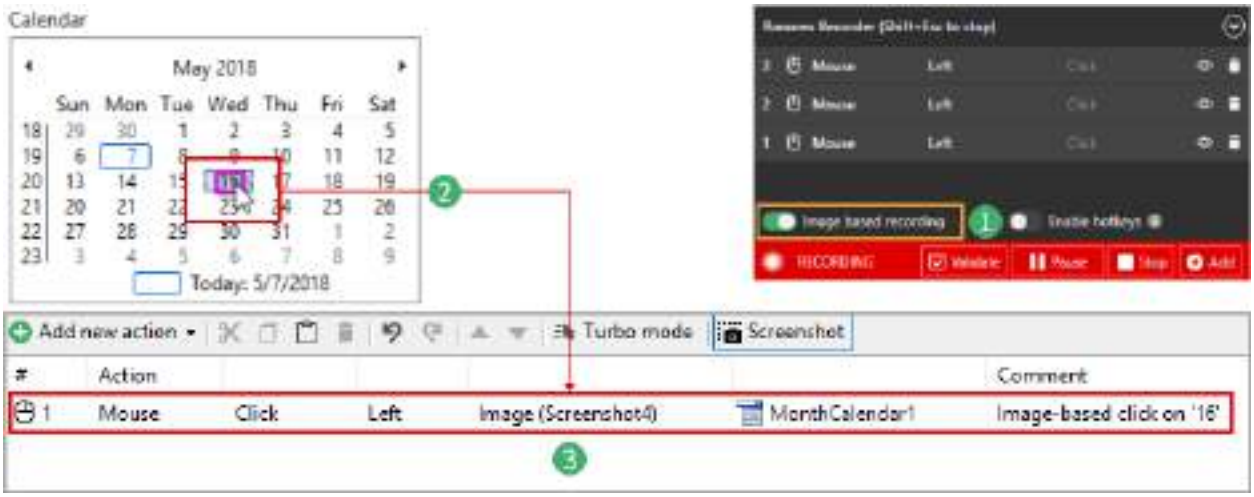
Sometimes minor changes, such as background or font color changes, can cause test failures. The challenge here is to ensure that UI elements are identified correctly despite these minor changes. Read on to find out how to do so.

Create the failing test

We'll first create the test so it fails for the above reason. Then, we'll fix it using the solution further below.

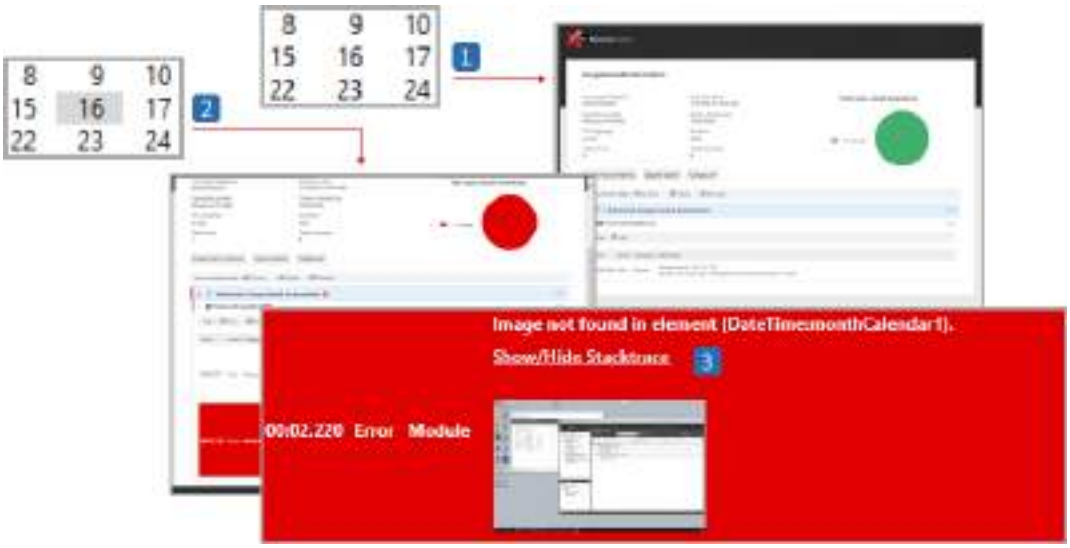
To create the failing test:

1. **Create** a new recording module, **start** recording and **turn on** image-based recording**.**
2. In the calendar view of the Ranorex Studio Demo Application, **click** a date and then **stop** recording.
3. **Check** the resulting action in your actions table.



Run the test

Now let's run the test and see when it's successful and when it fails.

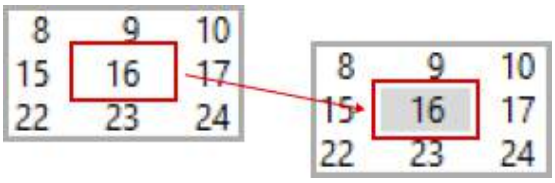


1 **Successful** test run

- When you run the test and the recorded calendar date is not selected in the demo app, the run succeeds.

2 **Failed** test run

- When you run the test and the recorded calendar date is selected in the demo app, the run fails.
- This is because the date's appearance changes slightly – **clicking it gives it a gray background.**
- This change is enough for Ranorex Studio to not recognize the element using the default image-based properties.



3 **Error message** in report

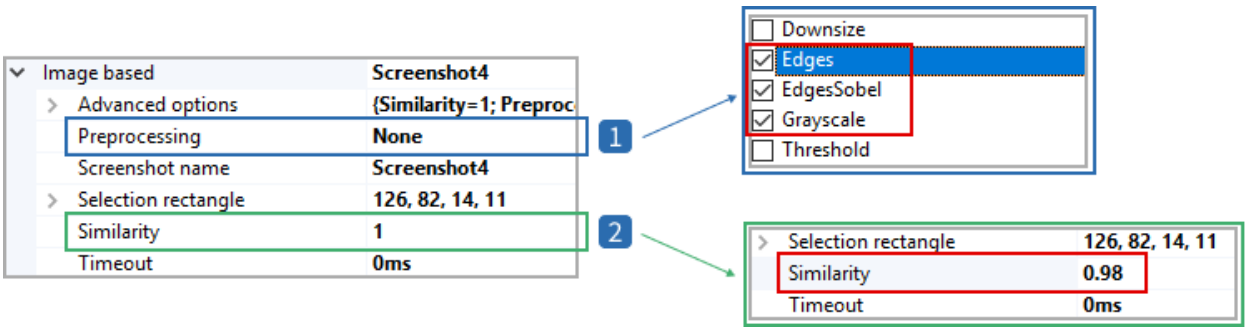
Solution

To solve this issue, we need to make changes to the image-based properties of the recorded action.

1 **Select** the action and **press** F4 to open its properties.

2 **Click Image-based.**

3 **Change** the following properties:



1 **Preprocessing**

- The preprocessing properties let you apply various filters to your comparison image.

- These filters can make image identification robust against changes in color, sharpness, brightness, and more.
- For our example, select the filters Edges, EdgesSobel, and Grayscale.

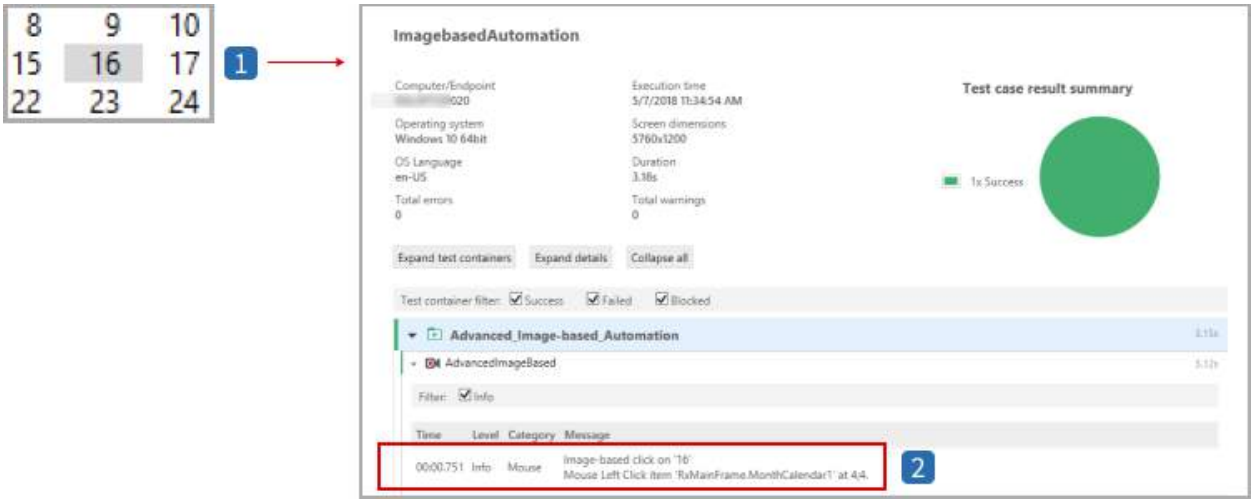
2 Similarity

- Preprocessing filters can remove a good deal of image differences, but not all of them.
- The similarity property lets you set an overall value of how similar the comparison image and the actual one need to be. The default is 1.0, i.e. 100 % or identical images.
- For our example, reduce the similarity to 0.98.

> **Further reading**
>
> For more information on preprocessing filters and similarity, refer to Ranorex Studio advanced > Image-based testing > ⇢ Image-based properties.

Rerun the test

Now rerun the test with the changed properties.



1 Modified calendar view with date and gray background.

2 Thanks to the changed properties, the date is identified correctly and the test run is successful.

Issue: Wrong UI element found

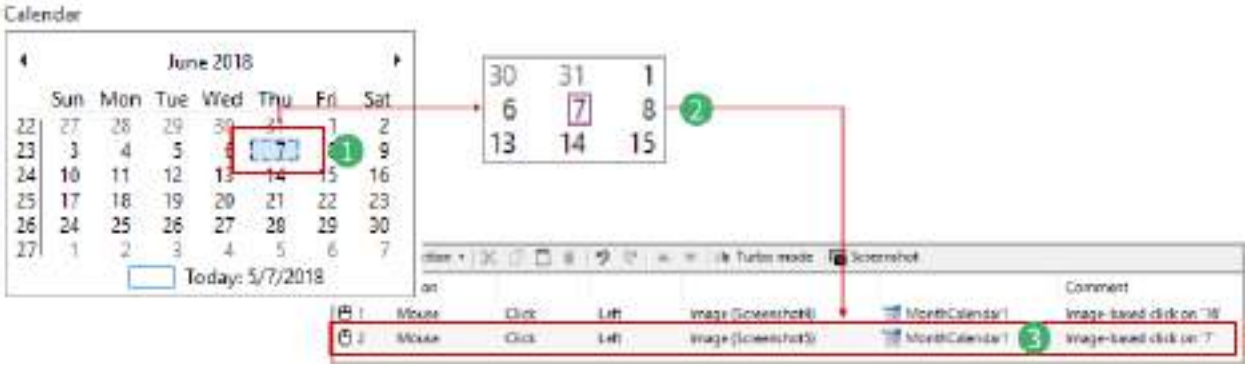
When running an image-based test, it may sometimes happen that Ranorex Studio doesn't find the recorded UI element, but a different, incorrect one that looks like the one recorded – a false positive. There are two ways to solve this issue. We'll show you both. We also recommend combining them for best results.

Create the failing test

We'll first create the test so it fails for the above reason. Then, we'll fix it using the solution further below.

To create the failing test:

1. **Start** a recording, **enable** image-based recording, and in the calendar view of the demo app, **click** a single-digit calendar date.
2. **Note** the size of the frame around the action spot, where the mouse click will be performed. This is the image-find region where Ranorex Studio will look for the 7. Its size is part of the issue. We'll fix this later as part of solution 1.
3. **Stop** the recording and **check** the resulting action in the actions table.



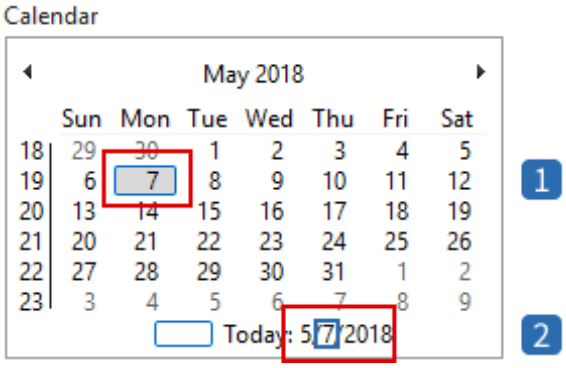
Run the test

Now let's run the test to see how it fails.

Note

For our example, we've intentionally also made sure the recorded date (7) is selected before the test run, so that it has a gray shade and looks different from what we recorded. This isn't necessary, but it ensures the test will find an incorrect 7 every time.

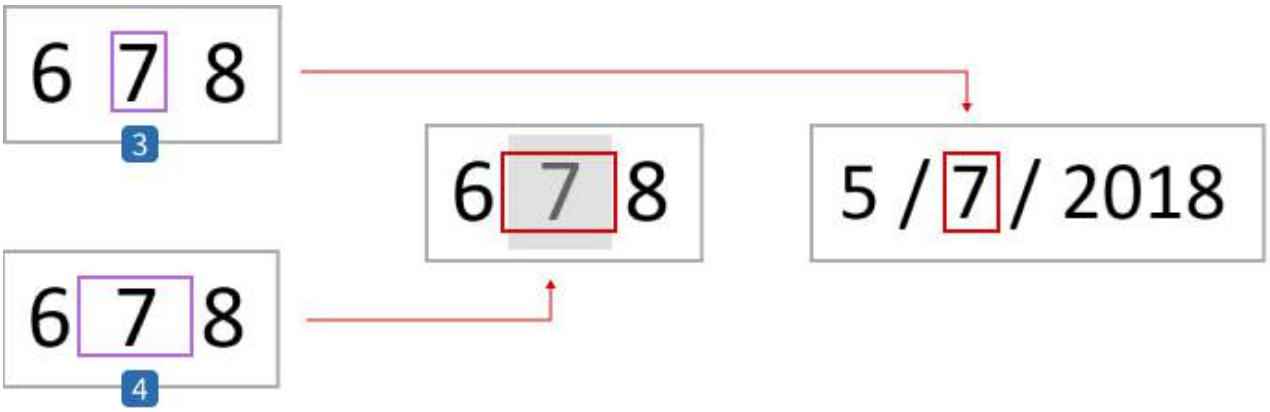
If you don't do this, the test may or may not find the correct 7, which is still unacceptable for a robust test. Give it a try yourself.



1. **June 7th** is the date we initially recorded. Compared to the recording it now has a gray background and therefore cannot be identified by Ranorex Studio (see note above).
2. In its search for a matching UI element, Ranorex Studio finds a different 7: the one in today's date below the calendar.

Solution #1

Remember the action spot we noted when creating the failing test? The image-find frame around it is too narrow here, i.e. doesn't contain enough information to make it unique. This is why it catches other 7s as well. We need to extend it using the built-in image editor.



3 **Frame of the default action spot**: Surrounds the element very closely. This is why it also catches similar elements.

4 **Extended action spot frame:** A wider frame includes more information (even if it's just whitespace here) and is therefore less likely to catch incorrect elements.

Change the frame in the image editor

To extend the frame, open the built-in ⇢ image editor:

1. **Select** the action and **click** the breadcrumb button next to the referenced image.
2. In the image editor toolbar, **click** the button **Select image find region** and **drag** an extended image-find region around the 7, including more whitespace than before (**zoom in** using the magnifying glass buttons if necessary).
3. **Click OK**.

The test should now only find the correct 7 when you run it again. As mentioned above, this is one way to solve the issue. Read on to find out about the second way, which works even better if you combine it with the first.

Solution #2

The second way to fix this issue makes use of the image-ignore region feature. Here, this feature lets you define regions in the reference screenshot where Ranorex Studio won't look for the image feature defined in the image-find region.

This way, we can simply ignore the part of the image that causes a false positive: the bottom part showing today's date. To do so:

1. In the ⇢ image editor, **click** the button **Select image ignore region**.
2. **Drag** a frame around the section showing today's date.

Now when you run the test, Ranorex Studio will ignore this section and not look for the action spot/the image-find region there, preventing a false positive.

Tips for image-based testing

Image-based testing can often require lots of experimentation and a good deal of experience. Follow these tips to make your life easier:

Uniqueness

The images you want to identify should be as unique as possible, with no, or few, alternative matches. Also use image-ignore regions to exclude similar features and prevent false positives.

Size

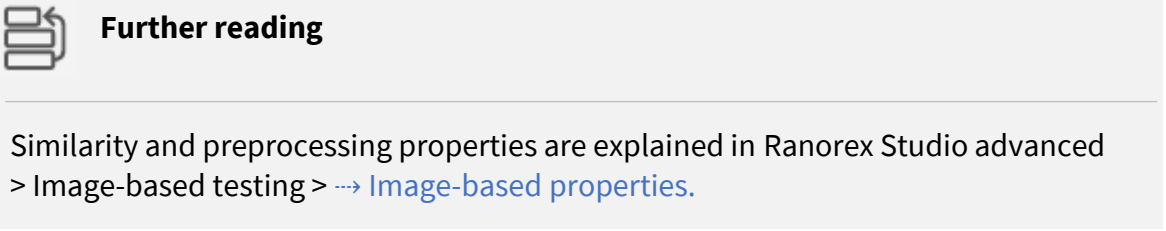
Make your comparison image as small as possible and as large as necessary. This may require some testing and experimenting.

Similarity

A similarity of 1.0 will only find an identical image. This is often counterproductive. A similarity of 0.99 is usually a better idea. Sometimes, you may need to go lower, but you should rarely go beyond 0.95. At this similarity value, images can already differ significantly.

Preprocessing filters

Use preprocessing filters! They can make your test more robust against changes in color, detail, and brightness. For example, if you have a UI element with a unique shape that may change its color due to input, you can use grayscale and the Edge filters. This way, Ranorex Studio will still find it despite color changes.



Further reading

Similarity and preprocessing properties are explained in Ranorex Studio advanced > Image-based testing > ⇢ Image-based properties.

Image editor

When using image-based testing, it's often necessary or useful to edit the captured screenshots for better results. Ranorex Studio has a simple built-in image editor for this purpose. On this page, we'll explain how it works and what functions it offers.

Start the image editor

To start the image editor:

1. In the actions table, **click** the breadcrumbs **...** next to a screenshot.
2. The image editor opens, displaying the screenshot.

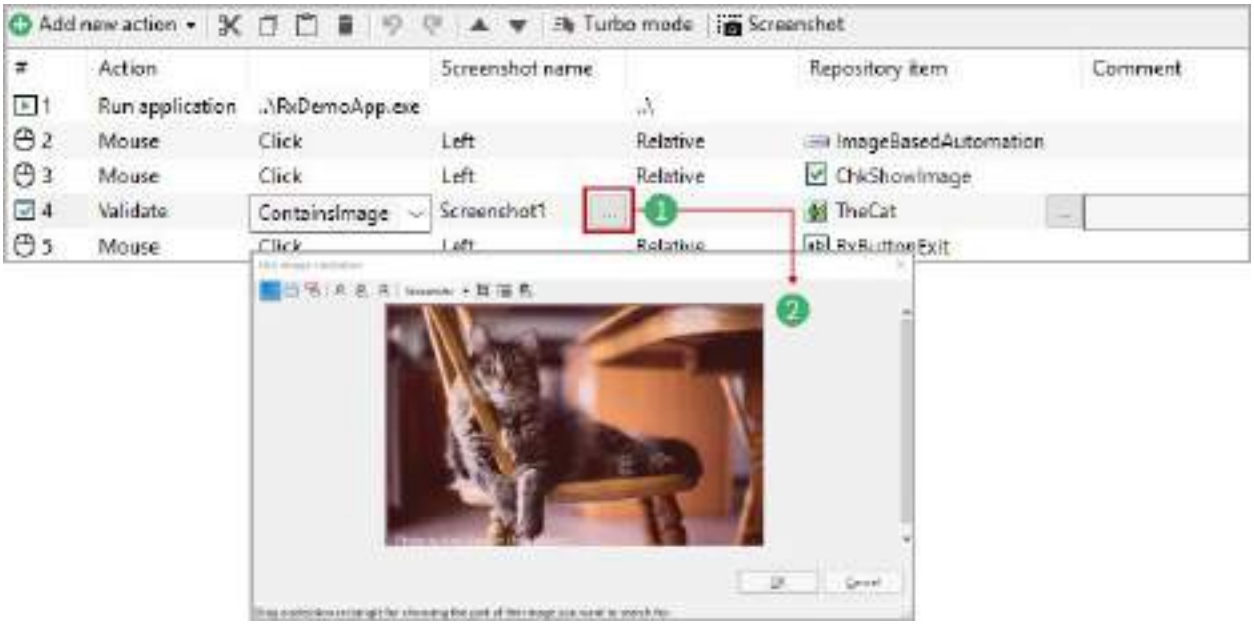


Image editor functions

1. **Select image find region**: Select a region in the image by dragging a rectangle. This region is what Ranorex Studio will try to find during test execution.
2. **Select image ignore region:** Select one or more regions by dragging a rectangle. Ranorex Studio will ignore these regions when trying to find the image find region during test execution.

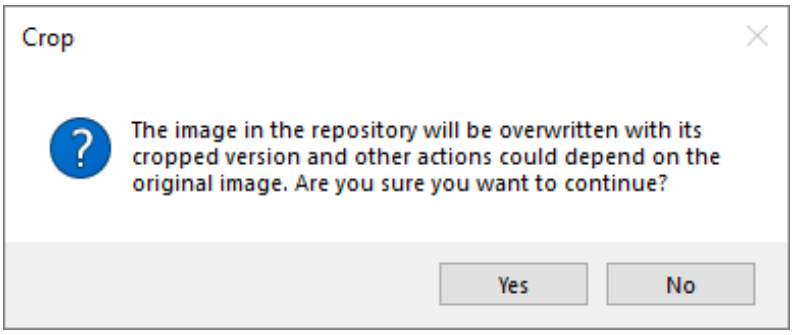
> **Note**
>
> This function **works differently** depending on what mechanism Ranorex Studio uses in an action to identify an image-find region during execution. There are two mechanisms: **CompareImage** and **ContainsImage**. They are based on the available image-based validations in Ranorex Studio.
>
> All non-validation actions (Mouse click, Key sequence, etc.) use ContainsImage. Validations can use one or the other.
>
> In short: For ContainsImage, image-find and image-ignore regions should not overlap (likely test failure). For CompareImage, the image-ignore region must be inside the image-find region. For more details, refer to CompareImage/ContainsImage under Validation in ⇢ Action properties.

3. **Autoselect image find region**: Click an area to auto-select a viable image find region.
4. **Zoom in/out**: Lets you zoom in and out. Press the middle button to reset the zoom to 100 %.
5. **Crop function**: First click the Select image find region button and specify a region. Then click the crop button to crop the image to that region.

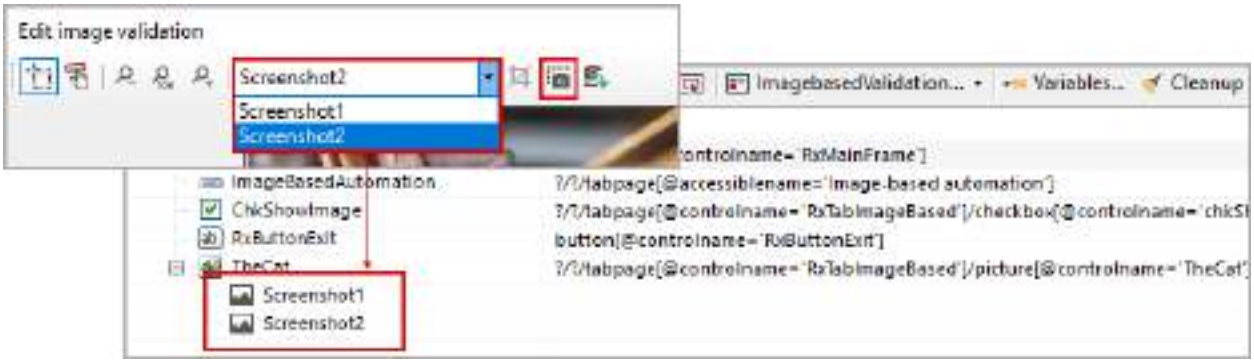
> **Note**
>
> A single image may be used for several validations. Therefore, be careful with cropping an image because you may lose important validation information.

Attention

Cropping overwrites the previous image. If you used the image in other actions as well, they may not work correctly with the new cropped image.



6 **New screenshot**: Lets you add another screenshot. Useful if something changed in the UI. When adding a new screenshot, it will be stored under the respective repository item along with the previous screenshot. Use the drop-down menu to choose between available screenshots.



7 **Open screenshot from file**: Same as adding a new screenshot, except that you browse to an existing file on your computer.

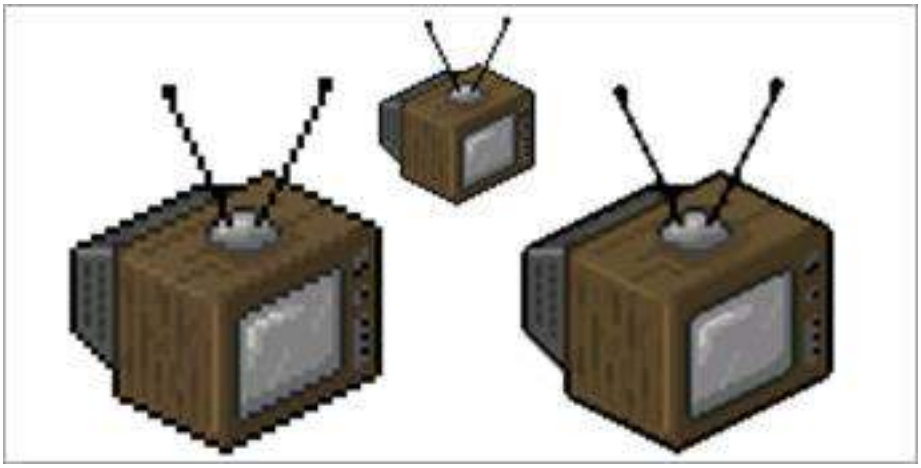
Image-based properties

On this page, we explain the preprocessing filters and the similarity value in more detail. For how to apply them, please refer to the first issue in ⇢ Advanced image-based testing.

Downsize

Downsizing means reducing the scale of an image. With vector graphics, this causes no loss of quality. However, with raster graphics (such as screenshots), downsizing means reducing the number of pixels. Therefore, quality is lost.

The downsizing filter's purpose is to reduce an image in size while retaining its unique characteristics. This way, superficial differences are less likely to cause a test failure and test execution is sped up because fewer pixels need to be searched and compared.



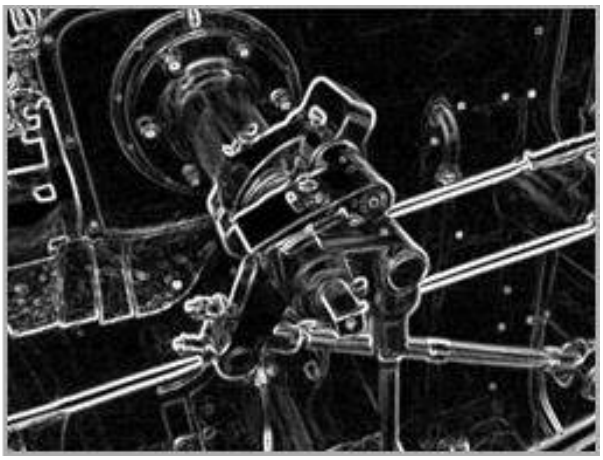
Edges

Edge detection is an image processing technique for finding the boundaries of objects within images. It works by detecting discontinuities in brightness.

The Edges filter reduces the amount of information in an image, which makes for faster test execution. It also makes image recognition more robust in terms of changes in color and brightness.

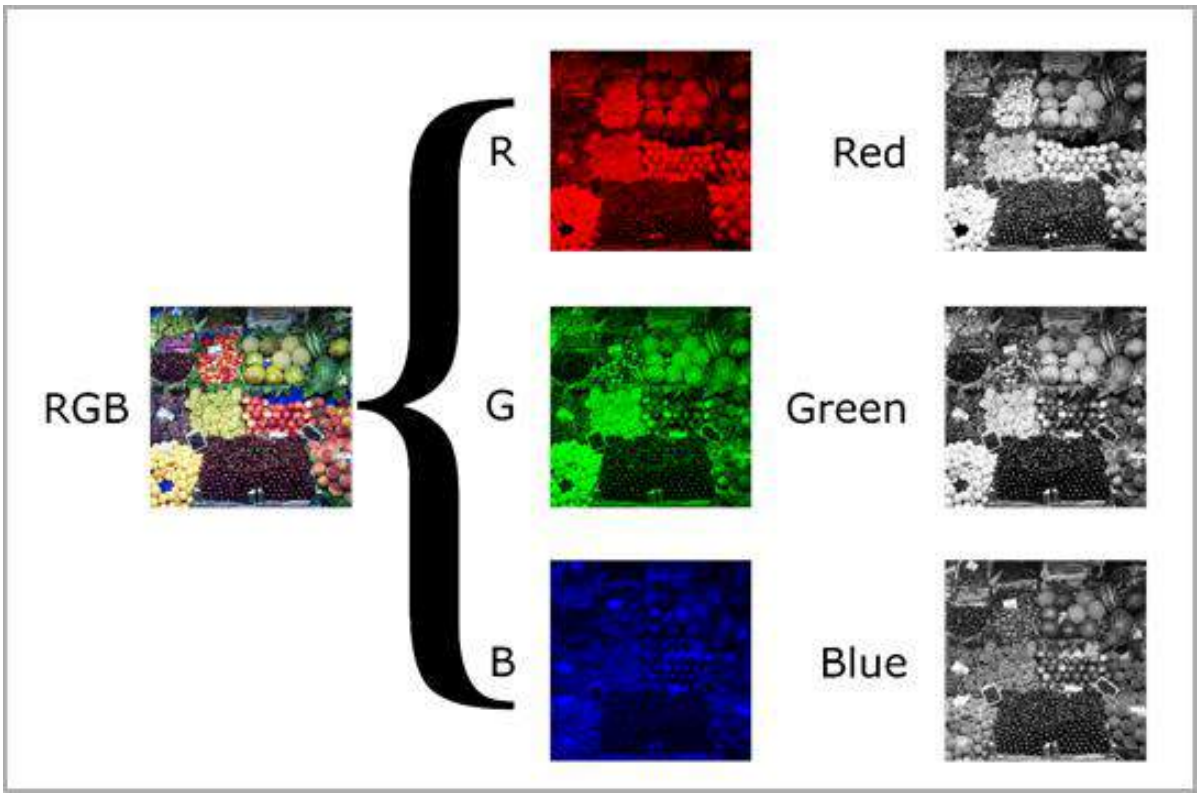
EdgesSobel

The EdgesSobel filter creates an image emphasizing edges. It's similar to the Edges filter and also makes image recognition more robust against against changes in color, brightness, and complexity.



Grayscale

The Grayscale filter reduces color and brightness information to 256 shades of gray, varying from black at the weakest intensity to white at the strongest. Grayscaling makes image recognition more robust in terms of changes in color.



Threshold

Thresholding is the simplest method of image segmentation. The simplest thresholding methods replace each pixel in an image with a black pixel if the image intensity is less than some fixed constant, or a white pixel if the image intensity is greater than that constant. From a grayscale image, thresholding can be used to create binary images. In the example image on the right below, this results in the dark tree becoming almost completely black, and the white snow becoming almost completely white.

The Threshold filter makes image recognition more robust against changes in color, detail, and brightness.

Similarity

The similarity property controls how similar the comparison and the actual image need to be for Ranorex Studio to consider them a match. It can be adjusted from 0.0 to 1.0. This corresponds to 0 % similarity (completely different images) and 100 % similarity (identical images). It may be tempting to use values like 0.8 or 0.9 to ensure the image is found even if some superficial changes occur. However, these values are only seemingly high. In reality, they are actually very low already.

At 0.9 similarity, Ranorex Studio would consider an entirely white 100-pixel picture identical to a picture with 90 white and 10 black pixels. That's quite a difference already. When you start comparing images in the magnitude of several thousand pixels, this becomes even more of an issue.

Similarity example #1

Consider the icons of Edge and Internet Explorer in the image below. They each have around 2000 pixels and are markedly different from each other. A 0.9 value would not catch these differences. It would consider them a match. In fact, you would need a minimum value of 0.95 for them to be treated as different.

For this reason, we recommend you always use as high a similarity value as possible. If 1.0 doesn't work, 0.9999, 0.99, or 0,98 should normally be enough. You should rarely go below 0.95, as this will ignore significant differences. To ensure your images are found at these high values, make sure to use **uncompressed image formats**, such as .png and .bmp. The artifacts created during compression make formats like .jpg unsuitable.

For large pictures in the order of several thousand pixels and more, we also recommend you turn off similarity reporting, as it can take a very long time to compute even on fast machines.

Similarity example #2

Similarity defines (as a percentage) **how similar** the compared images need to be in order to pass the check. The **color differences** for each pixel are calculated and the **mean squared differences are summed up**.

Example:

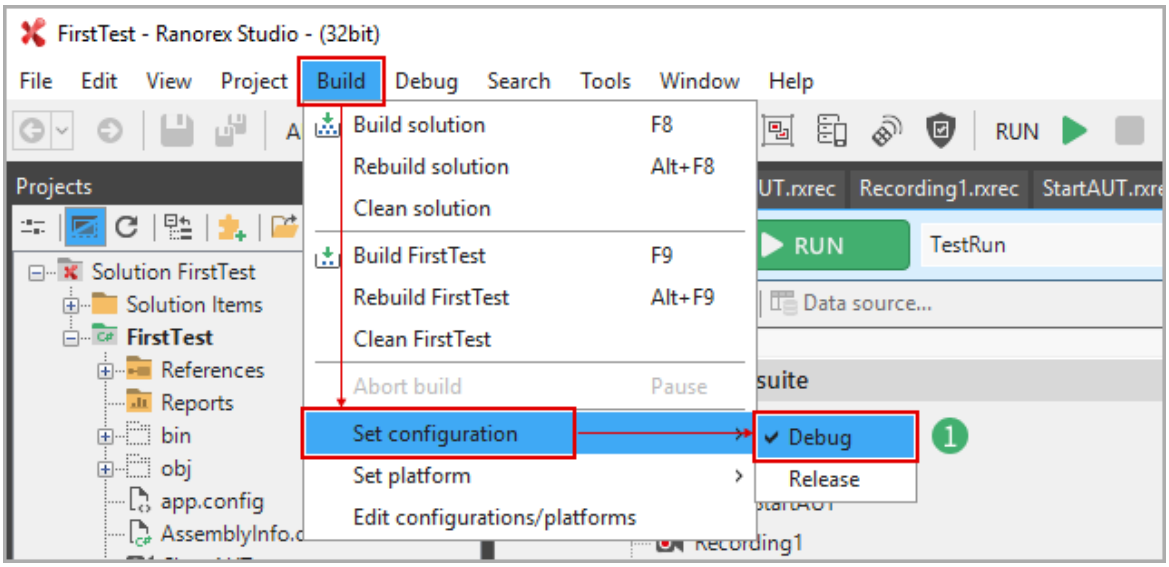
- Imagine that we compare **10×10 pixel** color images
- If all pixels have the same color except for **one pixel** being white (RGB 255,255,255) in picture A and black (RGB 0,0,0) in picture B, then the similarity is 99%
- If all pixels have the same color except for **one pixel** being black in pic A and one being gray (RGB 128,128,128, i.e. 50% color difference) in pic B, then the similarity is 99,75%
  (because of the squared error)
- Simply speaking, a similarity of 99% is already a quite low similarity if you compare large images and want to find small differences.

Maintenance mode

Maintenance mode allows you to catch and resolve certain errors during test execution. You can then apply the fixes to your test suite from the report. This saves time because you don't have to let the entire test run through to start diagnosing and fixing errors.

Activate maintenance mode

1. Under **Build > Set configuration**, **ensure Debug** is enabled. The option is usually enabled by default. It is not the same as the debug mode you enable from the menu bar.



2. In the test suite view, **click** the **Maintenance mode** switch to enable it.

Now when you run your test, it will be executed in maintenance mode. When you close the test suite view and reopen it, maintenance mode will be turned off again.

Note

Tests cannot be run in maintenance mode on a Ranorex Agent.

Catch and resolve errors

When maintenance mode is active, it catches errors native to Ranorex Studio and pauses test execution upon doing so. Depending on the error type, different dialogs with different options will appear.

Some options are always available:

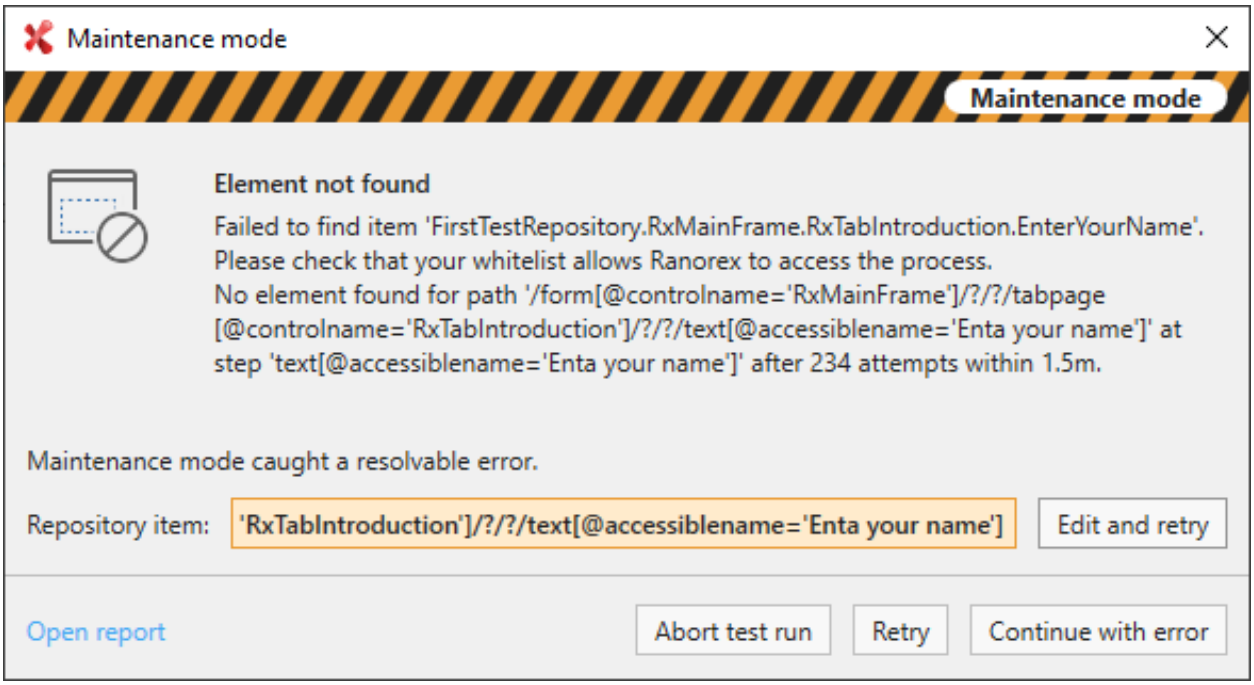
Open report	Opens the report.
Continue with error	Continues test execution according to the ⇢ error behavior set for the test container.
Abort test run	Aborts the test run.

Element not found

This dialog is displayed when a UI element couldn't be found. It displays the related error message and the RanoreXPath for the associated repository item.

Click **Edit and retry** to open Ranorex Spy. You can then retrack the element or change its path manually. Once you apply the changes in Spy, Ranorex Studio will automatically try to run the failed action again with the updated path.

You can also click **Retry** to try again without changing anything. This is useful if you think the path is correct and the element couldn't be found because the search timeout was too short or the AUT hadn't reached the correct state yet.



Validation failed

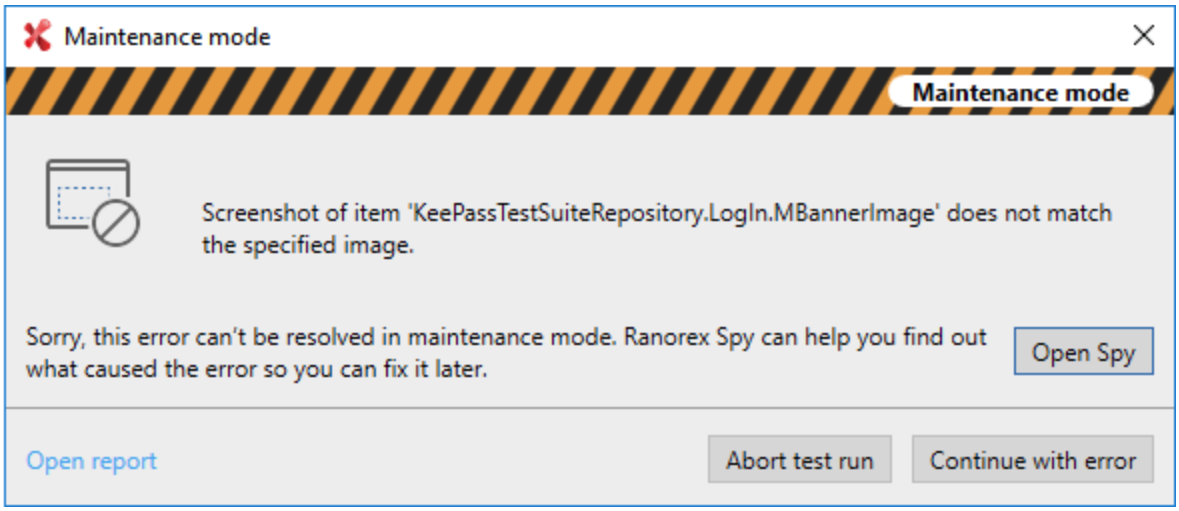
This dialog is displayed when an expected validation value didn't match the actual one. It displays the validation error message with the expected and actual values.

The drop-down menu contains the expected value (first option) and the actual value (second option). Select the actual value and then click **Apply and retry** to run the failed action again with this value.

> **Note**
>
> Image validations cannot be resolved in maintenance mode. The generic dialog will be displayed for them.

Generic dialog

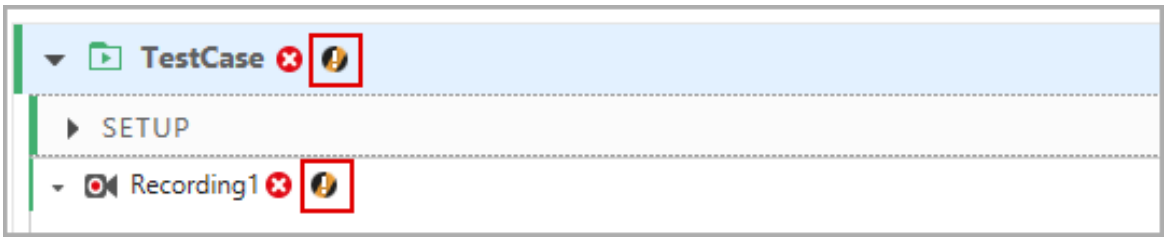
This dialog appears for all Ranorex-Studio-native errors that are currently not resolvable in maintenance mode. Click **Open Spy** to inspect the affected element in Ranorex Spy. This can help you diagnose why the error occurred.



Apply your fixes

Once you've resolved the errors that appeared in maintenance mode and your test has executed successfully, you can apply the fixes to your test suite in the report.

In the report, maintenance mode events are indicated by a special icon:



1. **Expand** the report item to receive more information and **find** the maintenance mode event with the green check mark.
2. **Hover** over it and **click** the **Apply…** button to apply the fix to your test.

Note

Fixes for variables and data bindings cannot be applied automatically. You will need to manually fix these in the respective test container or data source. This applies to both validation values and RanoreXPaths.

Performance Tracing

Performance tracing is a useful feature for optimizing your test execution times. It collects detailed information about the time it takes events like repository item searches or mouse input actions to happen during a test run. It collects this information in a log that you can then view and organize with Excel, for example.

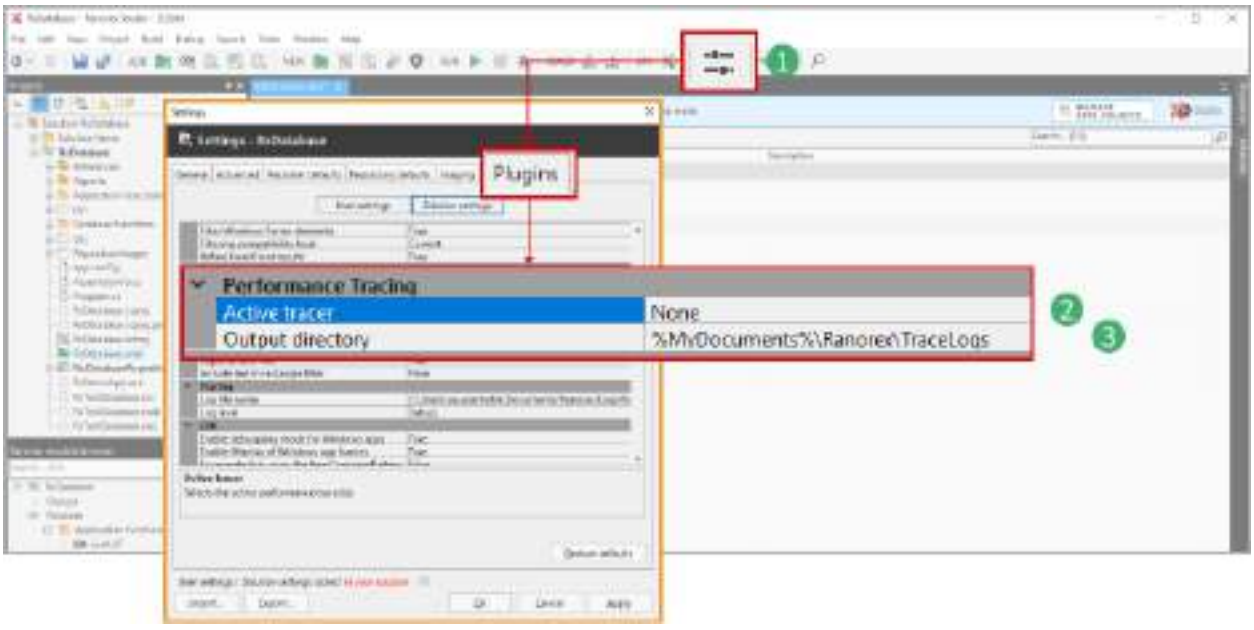
In this and the following chapters, you'll find out how to enable performance tracing, what the individual tracers collect, what the log files look like, and finally, some recommendations from us to get the most out of this feature.

Enable performance tracing

Performance tracing is disabled by default. When enabled, it applies to all test runs until you disable it again.

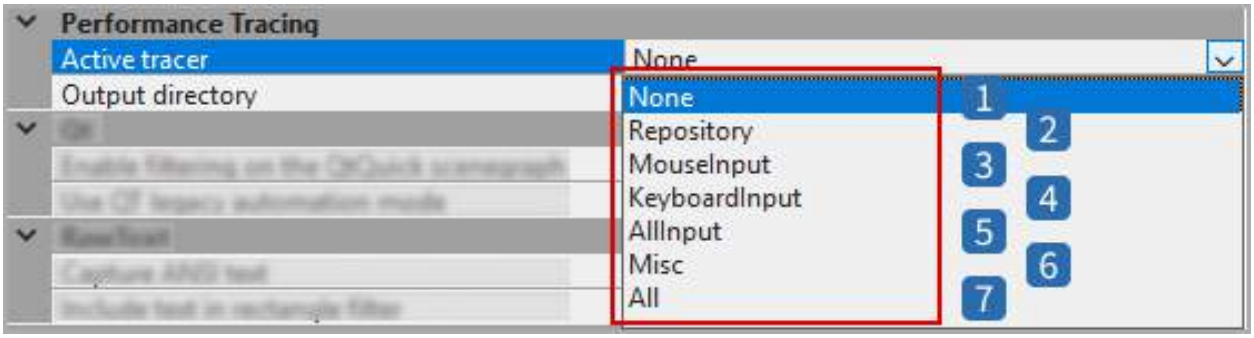
To enable it:

1. With a solution opened, **go** to **Settings > Plugins > Performance tracing.**
2. Under **Active tracer, select** a tracer (for descriptions see further below).
3. (optional) **Change** the default output directory for the generated trace logs. The default is **%MyDocuments%\Ranorex\TraceLogs**.



Tracers

There are 6 predefined tracers available. They allow you to customize what information you want to collect.



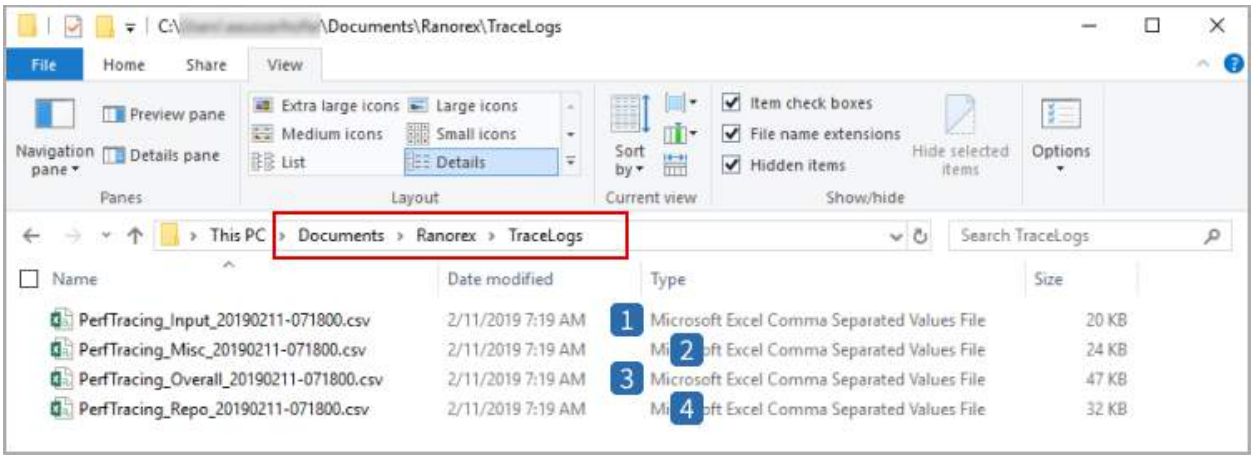
1. **None**
   Performance tracing is disabled. No information will be collected.
2. **Repository**
   Collects information about searched repository items (search time, RanoreXPath, search attempts, etc.)
3. **MouseInput**
   Collects information about mouse actions only (linked repository item, execution time, etc.). Applies to the following mouse actions: Move, Click
4. **KeyboardInput**
   Collects information about keyboard actions only (linked repository item, execution time, etc.). Applies to the following keyboard actions: Key sequence, Key shortcut press.
5. **AllInput**
   Combines the MouseInput and KeyboardInput tracers.
6. **Misc**
   Collects information about miscellaneous actions and processes, like fixed delays, tracing screenshots, and Run/Close application actions.
7. **All**
   Combines all of the other tracers. Collects the information in a more concise format.

Trace logs

In this chapter, we'll explain how trace logs generation works, what the trace logs look like when opened in Microsoft Excel (though you can open them in basically any spreadsheet or database program), and what the different headers in the files mean.

Trace log creation

Whenever you start a test run with any tracer active, Ranorex Studio will generate four timestamped logs (CSV format) in the configured output folder (default: **%MyDocuments%\Ranorex\TraceLogs**):



1. The log for all **Input** tracers
2. The log for the **Misc** tracer
3. The log for the **All** tracer
4. The log for the **Repository** tracer

During the test run, Ranorex Studio does the following:

- It checks which tracer is active and fills its log with the relevant data.
- It always also enters this data into the log for the All tracer in a more concise format.

For example, if the Repository tracer is active, Ranorex Studio fills the Repository log and the All log with data. The other logs remain empty.

When you start a new test run with a tracer active, this process is repeated with new timestamped logs.

Open trace logs

Ranorex Studio generates trace logs in the **CSV format** with commas as the delimiter. You can open them with virtually any spreadsheet or database program. We'll use Microsoft Excel for our explanations.

Normally, you shouldn't need to do anything for the logs to display correctly in a spreadsheet or database program. In some cases, however, you may need to import them manually and/or set the delimiter.

For example, all data may be in column A when you open a log in Microsoft Excel.

To fix this:

1. **Select** the entire column A.
2. In the menu bar, **click Data > Text to Columns**.
3. In the dialog that opens, **click Delimited > Next, check Comma, and click Finish.**

Trace log headers

The logs are created as CSV files with a header section. The headers specify what information was collected. The different logs have different headers, but they also share some. Below, you'll find an explanation of the headers in each log.

There are three headers that **you'll find in all logs**:

TimeStamp: The time an event (search for a repository item, action execution, etc.) started.

Module: The module the event occurred in.

TotalTime: The total time an event took in milliseconds.

Repository tracer

1. **RepoItem**
   The full name of the searched repository item.
2. **RepoItemId**
   The ID of the searched repository item.

3 **RepoItemPath**
The full RanoreXPath of the searched repository item.

4 **EffectiveTimeout**
The effective timeout in ms, as set in the properties of the repository item.

5 **TimesSearched**
How often Ranorex Studio attempted to find the repository item within the defined effective timeout.

6 **LastSearchTime**
How long the last successful search took in milliseconds.

7 **Found**
Whether the item was found (**TRUE**) or not (**FALSE**).

MouseInput, KeyboardInput, AllInput

1 **RepoItem**
The full name of the searched repository item.

2 **RepoItemId**
The ID of the searched repository item.

3 **Device**
Whether the action is a mouse or a keyboard action.

4 **InputType**
The specific type of mouse or keyboard input. This can be:

- for mouse: click, move
- for keyboard: sequence, press (Key shortcut press action in the recording module)

Other mouse/keyboard actions are practically instantaneous and therefore not collected.

5 **RepeatCount**
How often the action was performed.

6 **NominalDuration**
The nominal duration as set in the properties of the action.

Misc

1 EventType
The type of event that occurred. This can can be:

- fixed-delay: A delay action.
- datasource-load: Ranorex Studio loads a data source, e.g. an Excel file.
- proc-start: A Run application action.
- wait-proc-close: A Close application action.
- screenshot-trace: Tracing screenshots (Foreground/Background corresponds to the setting in the Report tab in the test suite properties).
- report-slow: Various reporting events that take longer than 10 ms.
- report-test-text-slow and report-test-data-slow: Ranorex Studio is busy creating the report and performance tracing is on hold.
- other: Events not native to Ranorex Studio, e.g. user code actions.

2 Details
More information relating to the EventType.

All

1 TestContainer
The test container the event occurred in.

2 **EventType**
The type of event that occurred. This can be:

- fixed-delay: A delay action.
- datasource-load: Ranorex Studio loads a data source, e.g. an Excel file.
- proc-start: A Run application action.
- wait-proc-close: A Close application action.
- screenshot-trace: Tracing screenshots (Foreground/Background corresponds to the setting in the Report tab in the test suite properties).
- report-slow: Various reporting events that take longer than 10 ms.
- report-test-text-slow and report-test-data-slow: Ranorex Studio is busy creating the report and performance tracing is on hold.
- other: Events not native to Ranorex Studio, e.g. user code actions.

3 **Details**
More information relating to the EventType.

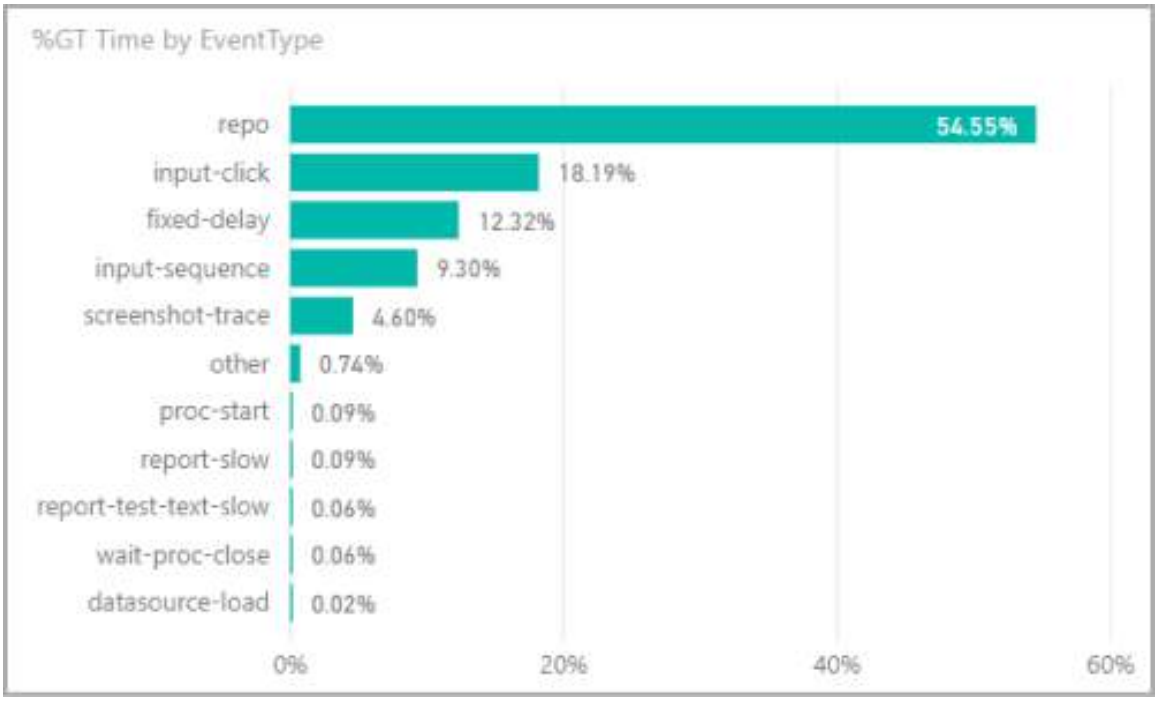
Leverage the data

Once you have your log files, it's time to evaluate the information they contain. This will show you where you can gain the most benefits from improving your test. In this chapter, we'll show you a few examples of evaluations that focus on typical reasons for unnecessarily long execution times.

Of course, we can't cover all scenarios, but these short examples can give you an idea on how to approach the wealth of information contained in the log files.

Time-by-event evaluation

A time-by-event evaluation is a good way of finding out which event types take the most time.

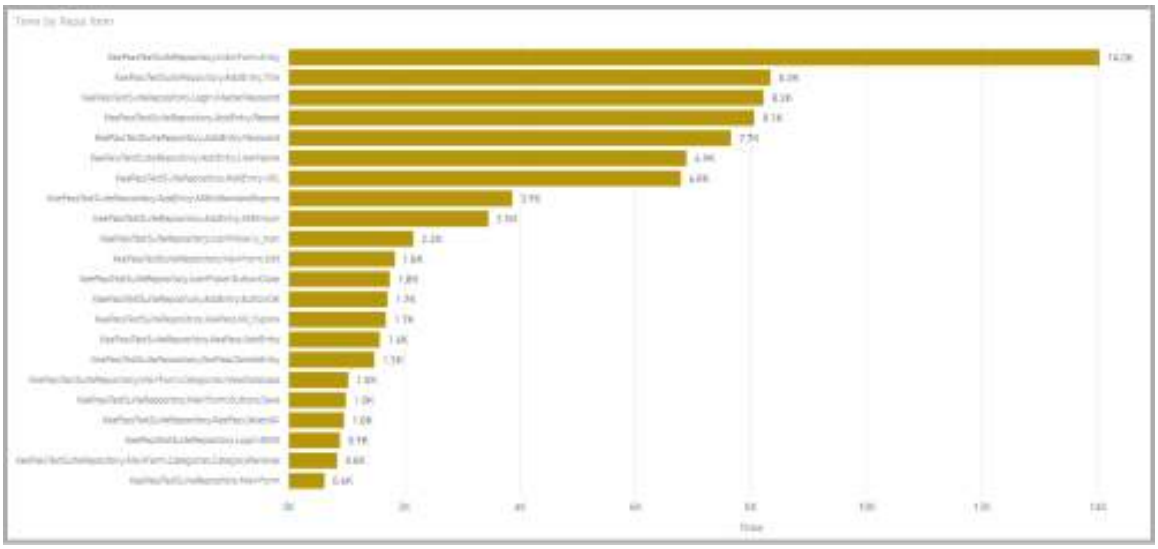


In the above example, we can see accessing repository items takes up half of the test execution time and mouse clicks and keyboard sequences take up another 30 % combined. Here, it may be a good idea to do another test run with the repository tracer and the all-input tracer to see in more detail which repository items and actions take up the most time, so you can optimize those, e.g. by adjusting the RanoreXPaths of repository items.

Delay actions also take up 12 %. It's a good idea to check out where you can replace those with Wait for actions, which only stop the test for as long as necessary instead of for a fixed amount of time.

Time-by-repository-item evaluation

Using the repository log, you can evaluate which repository items take the longest to be found and identified. This tells you which items you should take a closer look at, e.g. for adjusting the RanoreXPath to be quicker.



Time-by-test-container/module evaluation

The time it takes individual test containers or modules to execute is also a useful metric. Use it to find out where your test suite is slowest, so you can make structural improvements, for example, or remove clutter from test cases and modules.