Genero Ghost Client User Guide
Contents

Genero Ghost Client User Guide.................................................................5

What's New in 2.00.................................................................................5

Overview of Ghost Client and testing tools..............................................7

Install and configure.............................................................................8

The Genero Ghost Client framework......................................................8

The GGC directory................................................................................12

Templates..............................................................................................13
Examples..................................................................................................14
Example 1: check form (BDL).................................................................14
Example 2: check form (Java).................................................................15
Customizing a template...........................................................................15

Quick start with Ghost Client.................................................................17
Getting started.......................................................................................17
Deploy the price application on a Genero Application server.................20
Check failure.........................................................................................22
Scenario failure....................................................................................23

Testing with Ghost Client.....................................................................25
What you can test..................................................................................25
Limitations............................................................................................26
Test types..............................................................................................26
Write, compile, and run BDL tests.........................................................26
Write a Genero BDL test.......................................................................26
How to compile BDL tests.....................................................................28
How to run BDL tests...........................................................................28
Write, compile, and run Java tests.........................................................29
Write a Java test....................................................................................30
How to compile Java tests.....................................................................32
How to run Java tests...........................................................................32
Recording logs and generating scenarios..............................................33
Record a scenario log............................................................................34
Generate a scenario from a guilog.........................................................39
Generate a scenario from a GDC log....................................................39
Generate a scenario from a GBC log....................................................40
Set test environment................................................................. 40
Test speed ratio......................................................................... 41
Check certificates.......................................................................42
Authenticate with single sign on..............................................43
Front calls..................................................................................43
  Default front calls.................................................................43
  Register front call handler...................................................44
  Implement front calls in a test...............................................44
  Java front call parameters....................................................46
  Java front calls with BDL scenario........................................46
  Implement custom front calls...............................................47

**Genero Ghost Client reference**..............................................49
  Tools and Commands............................................................49
    gcadmin..............................................................................50
    ggcgen..............................................................................55
    GGC scenario program options.........................................57
    Default argument file........................................................59
  Display GGC version number................................................60
  The logging mechanism.........................................................61
    log.properties.....................................................................61
    Log files............................................................................62
  Genero BDL API for GGC.......................................................63
    Genero Ghost Client Types................................................73
    Actions and keys..................................................................78
    Form field interaction.......................................................79
    Table interaction..................................................................81
    Tree view interaction........................................................87
    Retrieve information and data.............................................89
    Retrieve MESSAGE and ERROR...........................................93
    Retrieve action state............................................................95
    Table information..............................................................96
    Session information............................................................98
    Front call request..............................................................102
    Front call answer................................................................105
  Java API functions for GGC..................................................110
    HTTP handlers..................................................................110
    GIPSSOHandler..................................................................111
    Status codes........................................................................111

**Demos**..................................................................................113
  Explore the scenario provider.................................................113

**Upgrading the GGC**.............................................................114
  GGC New Features................................................................114
    GGC 2.00 new features.........................................................114
    GGC 1.10 new features........................................................115
  Upgrade Guides for Genero Ghost Client...............................116
    GGC 2.00 upgrade guide.......................................................116

**Copyrights and Trademarks**..................................................119
Genero Ghost Client User Guide

Manual organization at a glance.

<table>
<thead>
<tr>
<th>General</th>
<th>Quick Start Guides</th>
<th>Testing</th>
</tr>
</thead>
<tbody>
<tr>
<td>• What's New</td>
<td>• Getting started on page 17</td>
<td>• What you can test on page 25</td>
</tr>
<tr>
<td>• Overview</td>
<td>• Check failure on page 22</td>
<td>• Test types on page 26</td>
</tr>
<tr>
<td>• Install and Configure</td>
<td>• Scenario failure on page 23</td>
<td>• The logging mechanism on page 61</td>
</tr>
<tr>
<td>• Framework</td>
<td></td>
<td>• Write, compile, and run BDL tests on page 26</td>
</tr>
<tr>
<td>• The GGC directory on page 12</td>
<td></td>
<td>• Write, compile, and run Java tests on page 29</td>
</tr>
<tr>
<td>• Templates on page 13</td>
<td></td>
<td>• Front calls on page 43</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Reference</th>
<th>Recording log files</th>
<th>Upgrading</th>
</tr>
</thead>
<tbody>
<tr>
<td>• Genero BDL API for GGC on page 63</td>
<td>• Record a scenario log on page 34</td>
<td>• GGC New Features on page 114</td>
</tr>
<tr>
<td>• ggcadmin on page 50</td>
<td>• Generate a scenario from a guilog on page 39</td>
<td>• Upgrade Guides for Genero Ghost Client on page 116</td>
</tr>
<tr>
<td>• ggcgen on page 55</td>
<td>• Generate a scenario from a GDC log on page 39</td>
<td></td>
</tr>
<tr>
<td>• GGC scenario program options on page 57</td>
<td></td>
<td></td>
</tr>
<tr>
<td>• Demos on page 113</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

For more in-depth documentation

Additional in-depth documentation is provided in the FGLGWS installation package for the Java API. For details of the Java packages and classes see the javadoc distributed with the Genero Ghost Client in your FGLGWS installation. The javadoc is found in your GGCDIR/doc/javadoc/ directory. See the help file by launching the /doc/javadoc/index.html file in your browser.

GGC 2.00 new features

A summary of new features and changes in functionality introduced with Genero Ghost Client 2.00.

Important: This page covers only those new features introduced with the Genero GGC version specified in the page title. Check prior new features pages if you migrate from an earlier version. Make sure to also read the upgrade guide corresponding to this Genero version.

Corresponding upgrade guide: GGC 2.00 upgrade guide on page 116.

Table 1: What's new in GGC 2.00

<table>
<thead>
<tr>
<th>Overview</th>
<th>Reference</th>
</tr>
</thead>
<tbody>
<tr>
<td>GGC is now a centralized backend to provide better load testing.</td>
<td>See The Genero Ghost Client framework on page 8.</td>
</tr>
<tr>
<td>A method is provided for multi-row selection.</td>
<td>See setRowSelection() on page 85</td>
</tr>
</tbody>
</table>
Methods are provided to retrieve column values in a matrix, table, tree, or screen record in your test scenarios and to return values for a column at a specified row.

Network load is simulated by loading all images attached to the AUI Tree. (Not done in TCP/Direct mode, only via GAS)

The GGC is embedded in the FGLGWS package in the $FGLDIR/testing_utilities/ggc directory.

Documentation about the APIs that make up the GGC framework can be accessed:

- For details of the Java packages and classes see the javadoc distributed with the Genero Ghost Client in your FGLGWS installation. The javadoc is found in your GGCDIR/doc/javadoc directory. See the help file by launching the /doc/javadoc/index.html file in your browser.
- For details of the BDL API, see Genero BDL API for GGC on page 63

Templates with snippets of generic code for testing are available in the installation directory. They are located in GGCDIR/template.

A feature of templates allows you to create a skeleton scenario with code from the header and footer templates. This provides the code that forms the basis of the test application.

<table>
<thead>
<tr>
<th>Table 2: Engine and Architecture</th>
</tr>
</thead>
<tbody>
<tr>
<td>Overview</td>
</tr>
<tr>
<td>The ggcadmin is an administration tool providing commands to start and stop the BDL scenario server, and to run Java scenarios in either direct or UA mode.</td>
</tr>
<tr>
<td>The ggcgen is the tool that you use to generate test scenarios. There are commands to generate BDL and Java scenarios.</td>
</tr>
<tr>
<td>The logging mechanism for the Genero Ghost Client allows you to control the level of logs and the output messages.</td>
</tr>
<tr>
<td>To ease Single Sign on detection, an HTTP handler named GIPSSOHandler is available.</td>
</tr>
<tr>
<td>Support for front calls is available.</td>
</tr>
<tr>
<td>GGC tools (ggcadmin, ggcgen and BDL scenarios) can load command options from a default argument file.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Table 3: Test generation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Overview</td>
</tr>
<tr>
<td>There is a debugging tool that is activated by the &quot;--dump_*&quot; options of the ggcgen tool.</td>
</tr>
</tbody>
</table>
The Genero Ghost Client (GGC) is a Java framework that allows you to test your applications. The GGC allows you to test the business logic and size the infrastructural needs of your applications. It acts as a *ghost client* because it does not render a graphical user interface. As a result, you can use it to test applications for different front-ends: Genero Browser Client (GBC), Genero Desktop Client (GDC), Genero Mobile for Android™ (GMA), or Genero Mobile for iOS (GMI).

For testing, you create different scenarios and user interactions on an application that the GGC can then run automatically against the application. The GGC can be used with a direct connection to the Dynamic Virtual Machine (DVM), or can use an application accessed via the Genero Application Server (GAS).

You can use the Genero Ghost Client to automate unit testing.

Test scenarios can be developed in two ways:

- Tests can be written by hand in either Genero Business Development Language (BDL) or Java.
- Test scenarios can be generated from the behavior described in a guilog file. The guilog file can be recorded using a direct connection or a GAS connection using any graphical client. The `ggcgen` on page 55 tool then generates Genero BDL files from the log file data, allowing the replay of the scenarios by the Genero Ghost Client.

**Important:** Use Genero BDL to create GGC applications. Genero applications will work for most unit and load tests. Java should only be used to perform critical load testing.

One of the key features of testing with the Genero Ghost Client is that you do not need to modify the original application code to write test scenarios.

The Genero Ghost Client allows you to develop working test case scenarios that can be run as required to test the stability of your applications before release.

**Related concepts**

- The Genero Ghost Client framework on page 8
- The Genero Ghost Client is a Java framework for building application tests.
- Testing with Ghost Client on page 25
Writing and running tests in Genero BDL and Java.

Install and configure for Genero Ghost Client

The Genero Ghost Client is embedded in the FGLGWS package. No additional installation is required, however configuration may be necessary.

Note: The Genero Ghost Client is also integrated with Genero Studio. See the Genero Studio User Guide for details.

The GGC is embedded in the FGLGWS package in the $FGLDIR/testing_utilities/ggc directory.

To use Genero Ghost Client API, you need a Java Development Kit (JDK). Make sure that your JDK version is at least version 8 or greater.

Configure for Genero tests

At a command prompt, navigate to FGLDIR and execute the script to set the environment.

Note: CLASSPATH, PATH, and FGLLDPATH are set for the GGC by the envcomp script

- On Linux®/UNIX™/macOS™ execute envcomp.sh
- On Windows®, execute envcomp.bat

Configure for Java

Set JAVA_HOME environment variable to the Java Development Kit (JDK) directory. Set PATH environment variable to include the JDK bin directory. For example, type one of the following sets of commands.

- On Linux®/UNIX™/macOS™:
  ```
  export JAVA_HOME=~/.Tools/java/jdk-11.0.1;
  export JDK_HOME=${JAVA_HOME};
  export PATH=${JAVA_HOME}/bin:${PATH}
  ```
- On Windows®:
  ```
  set JAVA_HOME=C:\Program Files\Java\jdk-11.0.1
  set JDK_HOME=%JAVA_HOME%
  set PATH=%JAVA_HOME%\bin;%PATH%
  ```

The Genero Ghost Client framework

The Genero Ghost Client is a Java framework for building application tests.

GGC infrastructure overview

The GGC architecture is designed to test scenarios written in Genero Business Development Language (BDL) and Java. The underlying APIs define functionality that is independent of the implementation. It is made up of interfaces, classes, and tools that handle all of the low-level code of generating tests so that you can quickly build scenarios to test your applications.

Some of the main interfaces of the GGC are described in Table 5: Ghost Client main interfaces on page 9.
Table 5: Ghost Client main interfaces

<table>
<thead>
<tr>
<th>Interface</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SessionManager</td>
<td>Instantiates and manages Scenario instances according to incoming VM connections and new runtime launches.</td>
</tr>
<tr>
<td>Scenario</td>
<td>A Scenario is by default a function call handler. It invokes front calls and its Play method handles the interface with the client by providing the Client instance as a parameter.</td>
</tr>
<tr>
<td>ScenarioProvider</td>
<td>If there is more than one scenario, the SessionProvider is created and populated with the scenarios.</td>
</tr>
<tr>
<td>Client</td>
<td>The Client interface provides all the methods to interact with the DVM or get information on the current state or content of the application.</td>
</tr>
<tr>
<td></td>
<td>It includes methods for delays between actions, like wait(1000), generating events like action(&quot;quit&quot;), key(&quot;Tab&quot;), or collapseTree(&quot;myTreeView&quot;, 10 / row number /), and so on. It also provides some introspection utilities like getValue() that returns the value of the focused field.</td>
</tr>
</tbody>
</table>

For details of the Java packages and classes see the javadoc distributed with the Genero Ghost Client in your FGLGWS installation. The javadoc is found in your GGCDIR/doc/javadoc/ directory. See the help file by launching the /doc/javadoc/index.html file in your browser.

**BDL tests with Ghost Client**

The GCC supports tests using either:

- a TCP configuration (TCPConfig class) that uses a direct connection to the Dynamic Virtual Machine (DVM)
- a UA configuration (UAConfig class) that runs through the Genero Application Server (GAS) to the DVM

For example, if you are developing your scenarios using BDL, you can specify a UA configuration in the command line to run the test against the application running on the GAS:

```
fglrn test_sample ua --url http://localhost:6394/ua/r/myapp
```

For examples of running BDL tests, see How to run BDL tests on page 28.

Figure 1: Ghost Client BDL testing scenario using GAS on page 10 shows the BDL scenario server running three test scenarios through the GAS, with the ScenarioProvider managing the session.
The tests are implemented by the following GGC components:

**UAConfig/TCPConfig**

The UAConfig/TCPConfig is the configuration created for the session. In this case it is a UA configuration as the application is tested running on the GAS.

**SessionManager**

The SessionManager interface manages the runtime and the test Scenario. In this example, there is more than one scenario, so the SessionProvider is created and populated with the scenarios.

**GGC scenario program**

The scenario you generated for the test (test_sample in our example) includes all scenarios from the session. This program sends instructions to the BDL scenario server as each scenario is tested. There are instructions to select the scenario such as:

```java
CALL ggc.selectScenario(0)
```

These are passed in direct binding to the Java API to implement the interface.

### Java tests with Ghost Client

The GGC supports testing scenarios written in Java using either:

- a TCP configuration (TCPConfig class) that uses a direct connection to the Dynamic Virtual Machine (DVM)
• a UA configuration (UAConfig class) that runs through the Genero Application Server (GAS) to the DVM

For example, if you are developing your scenarios using Java you can specify a direct connection to the DVM (TCP) with the command line tool `ggcadmin tcp` to run the test in this mode:

```
ggcadmin tcp -w path/to/myapp -c "fglrun myapp" --scenario myTests.test_sample
```

For examples of running Java tests, see How to run Java tests on page 32.

**Figure 2: Ghost Client Java testing scenario using direct connection**

The tests are implemented by the following GGC components:

**UAConfig/TCPConfig**

The UAConfig/TCPconfig is the configuration created for the session. In this case it is a TCP configuration as the application is tested running in direct mode.

**SessionManager**

The SessionManager interface manages the runtime and the test Scenario. In this example, there is more than one scenario, so the SessionProvider is created and populated with the scenarios.
The GGC directory

The Genero Ghost Client directory contains samples programs, source files, templates, and documentation. The default location for the GGC is in the FGLGWS package in the `FGLDIR/testing_utilities/ggc` directory. The following directories are included.

Figure 3: `FGLDIR/testing_utilities/ggc`

Various files are included in the `ggc` root directory. It contains the `ggc.jar` file that provides the implementation of the GGC. These files will not require any modification.

- **bin**: Contains wrapper script files for running the command line tools `ggcadmin` on page 50 and `ggcgen` on page 55.
- **doc**: The `GGCDIR/doc/javadoc/` directory contains documentation detailing the Java packages and classes that make up the Genero Ghost Client. See the help file by launching the `/doc/javadoc/index.html` file in your browser.
- **jars**: Contains the Java jar files for the HTTP interface, logging, and encoding.
- **lib**: Contains the `ggc.42m` BDL module that defines the BDL API.
src

Contains directories for the quick-start and the samples programs. The quick-start directory contains the source files for the Getting started on page 17 tutorial. It also contains the ggc-quick-start.gar for deploying to the GAS, and a build script for creating the Genero archive. The samples directory contains the demo directory with Java source files for the tutorial to Write a Java test on page 30. The ggc4lib directory contains ggc.4gl, source for the BDL API.

template

Contains directories for bdl and java, which contain a set of Templates on page 13 that are provided to generate tests. They contain files with the necessary front calls that characterize generic tests that perform checks, or execute actions (events). Corresponding files are either prefaced with check_, or events_.

Templates

The Genero Ghost Client has a set of built-in code templates.

Templates are fragments of code used by the GGC for generating scenarios. A template is an implementation of generic code that can work on different types of actions executed and checks that are performed based on parameters.

For instance, if the action you want to test was to sort a table column in your application form, the code snippet in the event_table_sort.4gl file can implement this action.

```
# Sort a table
CALL ggc.wait(${delayMs})
CALL ggc.sortTable("${tableName}", "${columnName}", ggc.${sortTypeBdl})
```

In the generated BDL test scenario the GGC engine passes the values for table, column, and sort selection to it:

```
# Sort a table
CALL ggc.wait(1910)
CALL ggc.sortTable("sr_prices", "name", ggc.SORT_ASCENDING)
```

There is a default set of templates provided in the installation directory. They are located in GGCDIR/template. There are templates for BDL and Java in the respective directories. These consist of the following:

- **event** type templates that generate code that trigger real actions in the program, like key events, set focus on a field, set a field value, sort a table, move a column, etc.
- **check** type templates that perform runtime checks on the state of the applications, like the current window name, field value, form name and title, and so on. All of these checks can be enabled by the ggcgen on page 55 command option --check-all, or they can be enabled individually, for example, --check-value.

**Why use templates?**

Templates allow you to customize the test implementation. If you update your templates, and regenerate your scenarios from the original log, the resulting scenario will take your modifications into account. This was not possible in GGC version 1.

As a simple example, if you have a form field called "today" containing today's date, the scenario in this case generates a check with the recorded value but that value can change when you execute the test at a later date.
Therefore, you may want to update the `check_value` snippet to ignore this field as it is expected that the check will fail on subsequent tests. Or you can implement a check for this field that ensures that it contains the date of the day. This allows you to customize the generated code to take cases like this into account without editing the test scenario code directly or having to record new test logs.

**Customizing templates**

You can develop your own custom templates and provide these to the `ggcgen` command option `--template-directory` to specify the custom template directory to use.

If customizing templates, the recommendation is to copy the templates you need from the `FGLDIR/testing_utilities/ggc/template` directory to your own working directory and make any changes needed there.

Multiple template directories can be provided to the `ggcgen` command option `--template-directory`. Lookup will be performed in the order of command line arguments, for example:

```
   ggcgen bdl --template-directory ${MYCUSTOM_TEMPLATE_DIR}
   --template-directory ${GGCDIR}/template/bdl/alternate-checks
   --check-all mylog.log
```

You will find the `alternate-checks` directory with templates using calls to `ggc.check*` functions, as an example of customized templates. These checks are a less-verbose form of the standard checks, but basically perform the same function. It is recommended to use these checks for your custom templates.

Templates are first searched for in the custom template directory, and then if not found in the default template directory.

**Ways the GGC uses templates**

Templates are used to generate a scenario for tests recorded in a `guilog` or GDC log file with the `ggcgen` on page 55 tool.

The scenario test file is either in BDL (`.4gl` file format) or Java (`.java` format), depending on what you specified in the `ggcgen` command option. It contains the template code expanded with parameter values corresponding to the action or event you performed when recording the log.

**Examples**

The Genero Ghost Client template examples.

The template examples in this section are found in the `GGCDIR/template` directory. To see more examples, see the `/bdl` and `/java` directories in this path.

Templates are snippets of code that are expanded with parameter values passed to them when generated. The syntax symbol `!!` (two exclamation marks) represent comments in the templates. These lines, which explain the function and the parameters, do not appear in the generated output.

If you are customizing your own templates, use the comment mechanism to document your template.

**Example 1: check form (BDL)**

The Genero Ghost Client BDL template to check form name and title.

```
check_form.4gl

!! Validates the current form name and title
!! This check is enabled if --check-all or --check-form parameter is used.
!!
!! Parameters:
```
Example 2: check form (Java)

The Genero Ghost Client Java template to check form name and title.

check_form.java

Customizing a template

You can develop your own custom templates and provide these to the ggcgen command to generate tests.

Before you begin:

The price application in the GGC/DIR/src/quick-start is already compiled and the log file price_sample.guilog has been generated. If you need to perform these tasks, see Getting started on page 17.

When customizing templates, the recommendation is to copy the templates you need from the FGLDIR/testing_utilities/ggc/template directory to your own working directory and make any changes needed.

About this task:

In this task, you customize the BDL check_form.4gl template for a specific test of a form name in the price application.

Steps

1. Create your custom template directory.
   This example uses a directory named /mycustom_template_dir.
2. Navigate to the FGLDIR/testing_utilities/ggc/template/bdl/alternate-checks directory.
   Copy the check_form.4gl template to your custom template directory
3. Edit the check_form.4gl file as shown.

Customizing a template

You can develop your own custom templates and provide these to the ggcgen command to generate tests.

Before you begin:

The price application in the GGC/DIR/src/quick-start is already compiled and the log file price_sample.guilog has been generated. If you need to perform these tasks, see Getting started on page 17.

When customizing templates, the recommendation is to copy the templates you need from the FGLDIR/testing_utilities/ggc/template directory to your own working directory and make any changes needed.

About this task:

In this task, you customize the BDL check_form.4gl template for a specific test of a form name in the price application.

Steps

1. Create your custom template directory.
   This example uses a directory named /mycustom_template_dir.
2. Navigate to the FGLDIR/testing_utilities/ggc/template/bdl/alternate-checks directory.
   Copy the check_form.4gl template to your custom template directory
3. Edit the check_form.4gl file as shown.
IF "$(formName)" == "price" THEN
  DISPLAY "It is the price form"
  # Ensure the form name is: $(formName)
  CALL ggc.checkFormName("$(formName)")
  # Ensure the form title is: $(formTitle)
  CALL ggc.checkFormTitle("$(formTitle)")
ELSE
  DISPLAY "Skipping form check on " || "$(formName)"
END IF

The template is modified to check for the "price" form only. The check is skipped for all other forms.

4. Generate the scenario from the guiLog.

    ggcgen bdl --template-directory path\to\custom_template_dir --check-form price_sample.guilog

Where:

• --template-directory option specifies the path to the custom template directory.
• --check-form parameter runs checks for the form name and title.
• The price_sample.guilog is used to generate the scenario.

The file price_sample.4gl is the created test scenario.

5. Compile the scenario.

    fglcomp price_sample.4gl

The generated test scenario compiles and creates the application file price_sample.42m.

6. Type one of the following commands to run the test.

   • UNIX® like OS:

     fglrun price_sample tcp --working-directory $(pwd) --command-line 'fglrun price'

   • Windows®:

     fglrun price_sample tcp --working-directory %CD% --command-line "fglrun price"

The test runs and the results display in the output. When the form "edit_price" is checked, "skipping check on edit_price" is displayed and there is no check to get its form name or title. This is in contrast to the "price" form check, as shown in this sample output, where getformname and getformtitle checks are performed. (Line breaks have been added to the output to improve readability.)

... 
skipping form check on edit-price
TCPClient.java:184 36 VM=1 "TCP Protocol"
   Notify interactive.. waitReady : false
 39 - "Send answer.." ("status":"SUCCESS","statusMsg":"Success","isFunctionCall":false)\n
Quick start with Ghost Client

Information to help you get started using Genero Ghost Client.

If you are a Genero Studio user, see the GUI Testing section in the Genero Studio User Guide.

Getting started

Prepare an environment for testing, compile a simple application, record a guilog, generate a scenario from the guilog file, and run tests.

About this task

In working with this procedure you get an understanding of Genero Ghost client basics and you see how you can use it for testing scenarios using your Genero BDL applications.

Steps

1. Set the environment:

   Source and run the environment script (envcomp or envcomp.bat) in your FGLGWS installation.
2. Build the sample application:

   **Tip:** The sample application (**price.xcf**) is already built when you installed the product - generally you can skip this step.

   • UNIX® like OS: navigate to the `quick-start` directory and run the build script:

     ```
     cd ${GGCDIR}/src/quick-start
     bash build.sh
     ```

   • Windows®: (you may have to copy the `quick-start` directory to a writable location before running the build script)

     ```
     xcopy /S "%GGCDIR%\src\quick-start" C:\tmp\quick-start
     cd C:\tmp\quick-start
     build.bat
     ```

     The **price** application is now compiled.

3. Start the GDC

   Launch the GDC monitor from the shortcut on the Start Menu. For more details for your OS, see the *Genero Desktop Client User Guide*.

4. Record a GUI log

   a) Run the **price** application using one of the following sets of commands.

   • UNIX® like OS:

     ```
     cd ${GGCDIR}/src/quick-start
     export FGLIMAGEPATH=${GGCDIR}/src/quick-start/img:${FGLDIR}/lib/
     image2font.txt
     fglrun --start-guilog=price_sample.guilog price
     ```

   • Windows®:

     ```
     cd %GGCDIR%/src/quick-start
     set FGLIMAGEPATH=%GGCDIR%/src/quick-start/img;%FGLDIR%/lib/
     image2font.txt
     fglrun --start-guilog=price_sample.guilog price
     ```

     The GDC opens the application in a system window.
Figure 4: Price application run by GDC

b) Execute some actions to record in the log before closing the application.

For example, sort the Product name or Price columns, edit values, etc.

The log file price_sample.guilog is generated by the DVM.

5. Generate a scenario from the GUI log.

The ggcgen bdl command transforms a DVM guilog into a BDL scenario.

```
ggcgen bdl price_sample.guilog
```

The file price_sample.4gl is the created test scenario.

6. Compile the scenario.

```
fglcomp price_sample.4gl
```

The generated test scenario compiles and creates the application file price_sample.42m.

7. Type one of the following commands to run the test.

- UNIX® like OS:
  ```
  fglrun price_sample tcp --working-directory $(pwd) --command-line 'fglrunch price'
  ```

- Windows®:
  ```
  fglrun price_sample tcp --working-directory %CD% --command-line "fglrun price"
  ```

The test runs and the results display in the output.

BDL Scenario server is not started.
Starting BDL scenario server on port 6500.
Listening on port 6500
BDL Scenario server not yet started, retry - Retry ..
BDL Scenario server not yet started, retry - Retry ..
SessionManager 0.001 ID=1 "SessionManager" Create.
SessionManager 0.002 ID=1 "SessionManager" Create TCP session.
tcp-d5d9cd930f5f4709966186945525f289 0.001 - "Initialize session"
FGLSERVER=localhost:36397
SessionManager 0.011 ID=1 "Session started" Session id: d5d9cd930f5f4709966186945525f289
SessionManager 6.099 ID=1 "Session ended" Session id: d5d9cd930f5f4709966186945525f289
SessionManager 6.100 ID=1 "Session ended" Notify .. idle
== Session statistics == Id: d5d9cd930f5f4709966186945525f289
  Duration:  
    Bytes received: 12213
    Bytes sent: 506
  Scenario count: 1
  Scenario failed: 0
SessionManager 6.101 ID=1 "Notify idle" -
SessionManager 6.102 ID=1 "Notify idle" done
Success

Now run the same test scenario using the Genero Application Server. See Deploy the price application on a Genero Application server on page 20.

**Deploy the price application on a Genero Application server**

**Before you begin**

Check you have the Genero archive ggc-quick-start.gar in your GGCDIR/src/quick-start directory. If not, run the build script described in Step 2.

**About this task**

In this task you try testing your scenario via the Genero Application Server. First you need to deploy the quick-start archive, and then you can run the test scenario you created earlier, this time using the standalone GAS.

**Steps**

1. **Set the environment for the GAS**
   
   Source and run the environment script (envas or envas.bat) in your GAS installation directory.

2. **Start the standalone dispatcher.**
   
   ```bash
   httpdispatch &
   ```

3. **Deploy the quick-start archive using one of the following methods:**
   
   - Log in to the deployment application at this URL:
     
     ```http://host:port/ua/r/admin_DeploymentApp```
   - Navigate to the Demos page: `http://localhost:6394/demos.html`. Select the **Deploy > Genero Archives** menu. Locate the quick-start archive file using the dialog. Then click **Deploy**.
   - Navigate to the quick-start directory and deploy the quick-start archive with the following `gasadmin` command:
     
     ```bash
gasadmin gar --deploy-archive ggc-quick-start.gar
     ```

   A successful deployment returns this output (output may differ slightly depending on the platform).

   ```bash
   Command succeeded.
   Found application price.xcf.
   Optimizing by compressing static resources...
   Optimizing by compressing public resources...
   Archive ggc-quick-start.gar successfully deployed.
   ```
4. Enable the quick-start archive using the deploymentApp or the command line:
   - In the deploymentApp select the **Deploy > Genero Archives ...** menu. In the Manage Genero Archives screen select the archive you wish to enable and click on the **Enable** button.
   - At the command line navigate to the quick-start directory and run the following `gasadmin` command:

     ```
     gasadmin gar --enable-archive ggc-quick-start.gar
     ```

     If the archive is enabled, the following output is displayed (output may differ slightly depending on the platform):

     ```
     Command succeeded.
     Install application price.xcf into C:\ProgramData\FourJs\gas\3.20.03-201902111027\app
     Archive ggc-quick-start.gar successfully enabled.
     ```

5. Ensure the application works correctly:
   - In a browser type the address:

     ```
     http://localhost:6394/ua/r/price
     ```

     You should see the "Product and prices" application open in your browser and be able to interact with it.

6. Finally, execute the compiled test scenario against the application running on the GAS.

   ```
   fglrun price_sample.42m ua --url http://localhost:6394/ua/r/price
   ```

   The test runs and the results display in the output.

   **Note:** The output is intended as a sample to show test session statistics, the format may change in future product releases.
Check failure

Test introducing a failed check on a form title.

Before you begin

The price application in the GGCDIR/src/quick-start is already compiled and the log file price_sample.guilog has been generated. If you need to perform these tasks, see Getting started on page 17.

About this task

In this task you are simulating a typical development situation. The development team have changed the title of a form in the app and this is trapped in your GGC test.

Steps

1. Start the GDC
   
   Launch the GDC monitor from the shortcut on the Start Menu. For more details for your OS, see the Genero Desktop Client User Guide.

2. Generate a test scenario from the GUI log.

   ```
   ggcgen bdl --check-form price_sample.guilog
   ```

   In this example the ggcgen bdl command is run with the --check-form option just to check the form title.

   The file price_sample.4gl is the created test scenario.

3. Modify the form name check.
   
   Open the price_sample.4gl for editing. Change the form name check in the original code from "price" to "bad price". For example, in the original code you have:

   ```
   # Ensure the form name is: price
   CALL ggc.assert(
       ggc.getFormName() == "price",
       SFMT("Form name is not valid, expected: 'price', got: '%1'",
        getFormName()))
   ```

   Update this with:

   ```
   # Ensure the form name is: bad price
   CALL ggc.assert(
       ggc.getFormName() == "bad price",
       SFMT("Form name is not valid, expected: 'bad price', got: '%1'",
        getFormName()))
   ```

4. Compile the scenario.

   ```
   fglcomp price_sample.4gl
   ```

   The generated test scenario compiles and creates the application file price_sample.42m.

5. Type one of the following commands to run the test.

   • UNIX® like OS:

   ```
   fglrun price_sample tcp --working-directory $(pwd) --command-line 'fglrune price'
   ```
Windows®:

```bash
fglrun price_sample tcp --working-directory %CD% --command-line "fglrun price"
```

The test runs and the results are displayed in the output.

**Note:** The output is intended as a sample to show test session statistics, the format may change in future product releases.

BDL Scenario server is not started.  
Starting BDL scenario server on port 6500.  
Listening on port 6500  
BDL Scenario server not yet started, retry - Retry ..  
BDL Scenario server not yet started, retry - Retry ..  
SessionManager 0.000 ID=1 "SessionManager" Create.  
SessionManager 0.003 ID=1 "SessionManager" Create TCP session.  
tcp-1d60e45df7e34af3a01685dce25cd7c3 0.000 - "Initialize session"  
FGLSERVER=localhost:31339  
SessionManager 0.011 ID=1 "Session started" Session id:  
1d60e45df7e34af3a01685dce25cd7c3  
== Session statistics == Id: 1d60e45df7e34af3a01685dce25cd7c3  
Duration:  
  Bytes received: 12213  
  Bytes sent: 506  
  Scenario count: 1  
  Scenario failed: 0  
Check failures:  
/opt/fourjs/fglgws/testing_utilities/ggc/src/quick-start/price_sample.4gl::info:(GGC-2) -  
  Form name is not valid, expected: 'bad price', got: 'price'  
SessionManager 6.150 ID=1 "Session ended" Session id:  
1d60e45df7e34af3a01685dce25cd7c3  
SessionManager 6.150 ID=1 "Session ended" Notify .. idle  
SessionManager 6.151 ID=1 "Notify idle" -  
SessionManager 6.151 ID=1 "Notify idle" done  

The scenario is not marked as failed, but notice that there is a Check failures section, and that there you see "GGC-2" with the message describing the failure.

**Related concepts**  
Statistics on page 76  
The statistics type defines a record for retrieving test result statistics.

### Scenario failure

Test introducing a failed scenario.

**Before you begin**

The price application in the GGCDIR/src/quick-start is already compiled, the price_sample.guilog file has been generated, and the test scenario price_sample.4gl exists. If you need to perform these tasks, see Getting started on page 17.

**About this task**

In this task you are testing for a scenario failure by trying to trigger an action in the application that does not exist. This is trapped in your GGC test.

**Steps**

1. Start the GDC
Launch the GDC monitor from the shortcut on the Start Menu. For more details for your OS, see the *Genero Desktop Client User Guide*.  
2. Modify an action() call in price_sample.4gl.  
   Open the price_sample.4gl for editing. Replace an action name with "it_will_fail". For instance, in the original code you have:

```
# # Trigger action accept
CALL ggc.wait(0)
CALL ggc.action("accept") -- OK
```

Update this with:

```
# Trigger action it_will_fail
CALL ggc.wait(0)
CALL ggc.action("it_will_fail") -- OK
```

3. Compile the scenario.

```
fglcomp price_sample.4gl
```

The generated test scenario compiles and creates the application file price_sample.42m.  
4. Type one of the following commands to run the test.
   - UNIX® like OS:

```
fglrun price_sample tcp --working-directory $(pwd) --command-line 'fglrun price'
```

   - Windows®:

```
fglrun price_sample tcp --working-directory %CD% --command-line "fglrun price"
```

The test runs and the results are displayed in the output.

**Note:** The output is intended as a sample to show test session statistics, the format may change in future product releases.
Testing with Ghost Client

Writing and running tests in Genero BDL and Java.

What you can test

Any interaction a user would normally do on an application can be automatically tested by the Genero Ghost Client. Functions that perform actions can be tested once to see if they work as expected. You can then run them at any time during development to continually verify that changes to the application do not affect previously tested features.

For testing purposes it may be helpful to focus on building tests that fall into two categories: introspecting or checking the application AUI tree or executing events (user actions).

Introspecting the application AUI tree

These are some examples of introspecting the AUI tree:

- List available menu actions
- List available fields
- Get value of a given field
- Get the field that has the focus

Executing a user action

These are some examples of executing a user action:

- Send an action
- Input a value in a field
- Scroll through a table
- Open a tree view node

While most user actions are supported by the Genero Ghost Client (GCC), there are features that are not supported. See the Limitations on page 26 topic.

Related reference

Genero BDL API for GGC on page 63
The Genero BDL API for the GGC provides types, functions, and methods for generating or writing tests in Genero BDL.

**Limitations**

There are some limitations in what you can test with the Genero Ghost Client (GGC).

It is recommended that you check the [Genero BDL API for GGC](#) on page 63 to make sure there is support for the action you wish to test. These are some test actions that are not supported:

<table>
<thead>
<tr>
<th>Parallel dialogs</th>
<th>Parallel dialogs are not supported in GGC tests.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Drag and Drop</td>
<td>The action of dragging and dropping an item in a test is not supported by the GGC.</td>
</tr>
</tbody>
</table>

**Related concepts**

[What you can test](#) on page 25

Any interaction a user would normally do on an application can be automatically tested by the Genero Ghost Client.

**Test types**

The Genero Ghost Client can assist with unit, load, and performance testing.

**Unit Tests**

*Unit testing* is a method of testing each feature of your application in isolation to make sure it works as expected.

A unit test should provide you with the anticipated responses to a given set of user input, showing that the feature is able to handle both correct and incorrect input. The following good practices are recommended when designing unit tests:

- Each 4gl application should have its own scenario.
- Make a complete list of the application features, from the smallest (for example, displaying the About... screen), to the biggest and / or the most important ones.
- For each feature in this list, write a single test that will test one (and only one) feature.
- If a primary feature implies a secondary feature (for example, to register a new customer, you need to fill out a form), the secondary feature's test should take place before the primary feature's test in the test sequence.

**Write, compile, and run BDL tests**

The Genero Ghost Client has different methods for creating tests depending on whether you are using Genero BDL or Java. In this section the topics are about writing, compiling, and running BDL tests.

For details of the methods that support the Genero Ghost Client BDL see [Genero BDL API for GGC](#) on page 63.

**Write a Genero BDL test**

To write a test in Genero BDL, one file may contain the function that manages the session and the function that runs tests.

**Steps**

1. Begin by running the Genero Ghost Client ggcgen on page 55 tool to create a skeleton application.

```
  ggcgen bdl --skeleton myapp_tests
```
The `myapp_tests.4gl` file is created as shown:

```4gl
IMPORT FGL ggc

MAIN
    CALL ggc.setApplicationName("")
    CALL ggc.parseOptions()
    # Register scenario functions
    CALL ggc.registerScenario(FUNCTION play_0)
    # Start execution
    CALL ggc.play()
    EXIT PROGRAM 0
END MAIN

PRIVATE FUNCTION play_0()
END FUNCTION
```

In the sample code a `MAIN` block and a function called `play_0` is generated. The following describes the `MAIN` code statements:

- The FGL class `ggc` is imported.
- The call to the `ggc.setApplicationName` gets the application name to start.
- The call to the `ggc.parseOptions` method initializes the parse options that perform the following operations:
  
  a. Parse and validate command line options
  b. Initialize the scenario configuration
  c. Connect to the BDL scenario server
  d. Bootstrap the scenario
- The call to the `ggc.registerScenario(FUNCTION play_0)` method registers a scenario function for a test with the id of 0. A test may register multiple test scenarios. They run in the sequence registered.
- The call to the `ggc.play()` method starts the test of the registered scenarios.

2. To the `play_0` function add code to test an application.
In this simple test scenario the window name is retrieved from the client and the action cancel is triggered. In between these tests, there is a request for the client to "sleep" for the specified delay of 100 milliseconds.

Related concepts

Write, compile, and run BDL tests on page 26
The Genero Ghost Client has different methods for creating tests depending on whether you are using Genero BDL or Java. In this section the topics are about writing, compiling, and running BDL tests.

Demos on page 113
The Genero Ghost Client installation includes sample test scenarios for you to explore in the FGLDIR/testing_utilities/ggc/src/samples directory of your FGLGWS package.

Recording logs and generating scenarios on page 33
A recorded log file can be used by the Genero Ghost Client to store the scenario for testing. You can record logs for the purpose of generating scenarios for testing.

How to compile BDL tests
To compile your Genero BDL tests, you run the fglcomp tool command as shown from the command line.

Before you begin:

• It is assumed that you have created a Genero BDL test file, for example myapp_tests.4gl.
• It is assumed that you have placed the test file in the directory path/to/myTests.

To compile your test, at the command line type the following:

```
cd path/to/myTests ; fglcomp myapp_tests.4gl
```

An application (42m) file is created.

How to run BDL tests
The Genero BDL tests that you have compiled, are run from the command line as shown.

Before you begin:

• If you are using the GAS:
  • Make sure your GAS version is at least version 3.00 or greater.
  • Make sure that the standalone dispatcher httpdispatch is started and that you can access the GAS demos welcome page, http://localhost:6394/demos.html, from your browser. See the Genero Application Server User Guide for more information.

About this task:

When running tests, you need to pass the name of the application and how to test it as an argument. This can be either:

• a command (for direct mode)
• an URL (when running tests over GAS)

Examples of running tests for both methods are given.

Before you begin:
• It is assumed that you have compiled the Genero BDL test file, for example myapp_tests.4gl.
• It is assumed that you have placed the compiled file in the directory path/to/myTests and that you have changed to that directory.

Steps

1. Run the compiled test scenario against the application running on the GAS.

   \[ \text{fglr}\text{un myapp\_tests.42m ua --url http://localhost:6394/ua/r/myapp} \]

   The test runs and the results display in the output.

2. Type one of the following commands to run the test.
   • UNIX® like OS:
     \[ \text{fglr}\text{un myapp\_tests.42m tcp --working-directory $(pwd) --command-line 'fglr}\text{un myapp'} \]
   • Windows®:
     \[ \text{fglr}\text{un myapp\_tests.42m tcp --working-directory %CD% --command-line "fglr}\text{un myapp"} \]

   Where --working-directory is the option setting the path to the directory of the test application. In the example it is set to the current working directory.

   The test runs and the results display in the output.

Forward to GDC

The GGC does not render a graphical user interface. If you use forward-gui option, the GGC still connects to the DVM, but also forwards the Abstract User Interface (AUI) tree to the running Genero Desktop Client (GDC) so that you can see your scenario being executed. For example, to run the test through the GAS to view it running on the GDC on the localhost, run the following command:

\[ \text{fglr}\text{un myapp\_tests.42m ua --url http://localhost:6394/ua/r/myapp -f localhost:0} \]

Related concepts
GGC scenario program options on page 57
A generated GGC scenario implements a set of command line options.

Write, compile, and run Java tests

The Genero Ghost Client has different methods for creating tests depending on whether you are using Genero BDL or Java. In this section the topics are about writing, compiling, and running Java tests.

Note:

For details of the Java packages and classes see the javadoc distributed with the Genero Ghost Client in your FGLGWS installation. The javadoc is found in your GGCDIR/doc/javadoc/ directory. See the help file by launching the /doc/javadoc/index.html file in your browser.
Write a Java test

Write a simple test scenario, compile it, and execute it.

Before you begin:

- The tested application is the price application used in the Getting started on page 17.
- All source files are available in GGCDIR/src/samples directory.

1. Begin by running the Genero Ghost Client `ggcgen` on page 55 tool to create a skeleton application.

```
ggcgen java --skeleton myapp_tests
```

The `myapp_tests_provider.java` file is created in the "scenario" package:

```java
/*
 * Generated using Genero Ghost Client 2.00.06-201905141203
 */
package scenario;
import com.fourjs.ggc.ScenarioProvider;
import com.fourjs.ggc.Scenario;
import com.fourjs.ggc.Client;
import com.fourjs.ggc.generator.ScenarioChecks;
import java.util.concurrent.ConcurrentLinkedDeque;
public class myapp_test2_provider implements ScenarioProvider {
    private final ConcurrentLinkedDeque<Scenario> scenarios;
    /* Instance initializer */
    public myapp_test2_provider() {
        scenarios = new ConcurrentLinkedDeque<>();
        /* Register scenario functions */
        scenarios.add(new scenario_0());
    }
    /* Scenario myapp_test2 id: 0 */
    private class scenario_0 implements Scenario {
        @Override
        public void play(Client client) {
        }
    }
    @Override
    public Scenario nextScenario() {
        return scenarios.removeFirst();
    }
}
```

In this skeleton scenario:

- The code begins by importing the Genero Ghost Client Java classes.
- The `myapp_test2_provider()` method implements the `com.fourjs.ggc.ScenarioProvider` interface.
c. A function (scenario_0) with the id of 0 is registered in the scenarios variable. A test may register multiple test scenarios in this variable. They run in the sequence registered.

d. A scenario_0 class is declared. This implements the com.fourjs.ggc.Scenario interface, which requires the method play to run the test.

2. To the play method of the scenario_0 class add code to test an application.
For the test, the window name is retrieved from the client and the action cancel is triggered.

```java
package scenario;
import com.fourjs.ggc.ScenarioProvider;
import com.fourjs.ggc.Scenario;
import com.fourjs.ggc.Client;
import com.fourjs.ggc.generator.ScenarioChecks;
import java.util.concurrent.ConcurrentLinkedDeque;

public class myapp_test2_provider implements ScenarioProvider
{
    private final ConcurrentLinkedDeque<Scenario> scenarios;
    
    /* Instance initializer */
    public myapp_test2_provider()
    {
        scenarios = new ConcurrentLinkedDeque<>();

        /* Register scenario functions */
        scenarios.add(new scenario_0());
    }

    /* Scenario myapp_test2 id: 0 */
    private class scenario_0 implements Scenario
    {
        @Override
        public void play(Client client)
        {
            String windowTitle = client.getUserInterface().getCurrentWindow().getTitle();
            System.out.println("Window title is: " + windowTitle);
            System.out.println("Exiting with action 'cancel'");
            client.wait(100);
            client.action("cancel");
        }
    }
    
    @Override
    public Scenario nextScenario()
    {
        return scenarios.removeFirst();
    }
}
```

Related concepts

ggcadmin on page 50
The ggadmin is an administration tool providing commands to start and stop the BDL scenario server, and to run Java scenarios in either TCP direct or UA mode through the GAS.

**Recording logs and generating scenarios** on page 33

A recorded log file can be used by the Genero Ghost Client to store the scenario for testing. You can record logs for the purpose of generating scenarios for testing.

### How to compile Java tests

To compile your Java tests, you run the javac command as shown from the command line.

**Before you begin:**
- It is assumed that your Java test files are in the package `path.to.myTests` and therefore you have placed them in the directory `path/to/myTests`.
- It is assumed your environment is configured to run Java and the GGC, see Install and configure for Genero Ghost Client on page 8

**Steps**

To compile your tests, type the following:

```
javac path/to/myTests/*.java
```

### How to run Java tests

You run tests from the command line using the `ggadmin` tool which starts a testing session for an application.

**Before you begin:**
- It is assumed that your Java test files are compiled in the directory `path/to/myTests`
- If you are using the GAS:
  - Make sure your GAS version is at least version 3.00 or greater.
  - Make sure that the standalone dispatcher `httpdispatch` is started.
  - You must have deployed the app you are testing on the GAS. See Deploy the price application on a Genero Application server on page 20.

**About this task:**

In this task you use the `ggadmin tcp` command to run tests in direct mode, or `ggadmin ua` command to run tests against an application running on the GAS.

**Steps**

1. Set your CLASSPATH to point to your `path/to/myTests` directory.
   
   For example, run one of the following commands:
   - On Linux®/UNIX™/macOS™:
     ```
     export CLASSPATH=$CLASSPATH:/path/to
     ```
   - On Windows®:
     ```
     set CLASSPATH=%CLASSPATH%;/path/to
     ```

2. To run the tests through the GAS, perform the following:
   a) Run the application you wish to test.
      
      In your browser launch the application:
      ```
      http://localhost:6394/ua/r/myapp
      ```
b) Execute the scenario against the application.

```bash
ggcadmin ua --url http://localhost:6394/ua/r/myapp --scenario myTests.myScenario
```

Or alternatively you can run the test in direct connection mode

3. To execute the scenario against the application, perform the following.
   a) Use one of the following commands to execute the scenario against the application.

   - UNIX® like OS:
     ```bash
ggcadmin tcp -w path/to/myapp -c "fglrun myapp" --scenario myTests.myScenario
```

   - Windows®:
     ```bash
ggcadmin tcp -w "path/to/myapp" -c "fglrun myapp" --scenario myTests.myScenario
```

   Where `-w` is the option setting the path to the directory of the test application.

At the end of the scenario execution for both direct mode and GAS, statistics recorded during the session are displayed as shown in the sample output.

**Note:** The output is intended as a sample to show test session statistics, the format may change in future product releases.

Window title is: Product and prices
Exiting with action 'cancel'
...
== Session statistics ==
  Id: 981aeaaf9f8c4e33a0f67ef576f76393
  Duration: 0:00:01.370
  Bytes received: 10490
  Bytes sent: 146
  Scenario count: 1
  Scenario failed: 0
  Checks failed: 0

**Forward to GDC**

The GGC does not render a graphical user interface. If you use `forward-gui` option, the GGC still connects to the DVM, but also forwards the Abstract User Interface (AUI) tree to the running Genero Desktop Client (GDC) so that you can see your scenario being executed. For example, to run the test through the GAS to view it running on the GDC on the localhost, run the following command:

```bash
ggcadmin ua --url http://localhost:6394/ua/r/myapp --scenario myTests.myScenario -f localhost:0
```

**Related concepts**

ggcadmin on page 50

The ggcadmin is an administration tool providing commands to start and stop the BDL scenario server, and to run Java scenarios in either TCP direct or UA mode through the GAS.

**Recording logs and generating scenarios**

A recorded log file can be used by the Genero Ghost Client to store the scenario for testing. You can record logs for the purpose of generating scenarios for testing.

The topics in this section detail how to record a log file from the Dynamic Virtual Machine (DVM) and generate scenarios for testing.
Record a scenario log

Describes procedures used to record a log file of the interaction with an application launched to a Genero web client.

Record a guilog log using a direct connection

About this task:

In this task the --start-guilog option of the fglrun command is used to record a log file. For more information on using this command, see the GUI front-end connection chapter of the Genero Business Development Language User Guide.

1. Set the environment:
   Source and run the environment script (envcomp or envcomp.bat) in your FGLGWS installation.

2. Launch the Genero Desktop Client.

3. Start the application using fglrun and the --start-guilog option.
   For example:

   ```
   D:\fjs\gst\fgl\demo\Widgets>fglrun --start-guilog=D:/tmp/log/edit.log Edit
   ```

   In this example:

   • The DVM is run with the --start-guilog option to record a log file.
   • The interaction is saved to a file named edit.log.

   The GDC opens the application in a system window.

   ![Edit application](image)

   **Figure 5: Edit application**

4. Use the application to execute some actions to record in the log.

5. To finish recording, close the application.

When you have completed recording the log, your next task is to generate test sets from the recorded log file. See Generate a scenario from a guilog on page 39.
Record a scenario log via the GAS (DVM)

About this task:

In this task you record a log from the DVM end by adding the `fglrun --start-guilog` command in the DVM element of the application configuration file. When the application is run, the specified guilog file is generated.

1. Add a DVM element to the configuration file of your application.
   
   For example:

   ```xml
   <APPLICAITON Parent="defaultgwc" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xsi:noNamespaceSchemaLocation="http://www.4js.com/ns/gas/3.10/cfextwa.xsd">
   <EXECUTION>
   <PATH><path_to_your_application></PATH>
   <DVM>fglrun --start-guilog=/tmp/myapp.log</DVM>
   <MODULE>myapp</MODULE>
   </EXECUTION>
   <UA_OUTPUT>
   <PUBLIC_IMAGEPATH>$(res.public.resources)</PUBLIC_IMAGEPATH>
   <GBC>_default</GBC>
   </UA_OUTPUT>
   </APPLICATION>
   
   In this example:
   - The DVM is run with the `--start-guilog` option to record a log file.
   - The interaction with the application is saved to a file named `myapp.log`.

2. Save the configuration file in the `$(res.appdata.path)/app` directory.

3. Set the environment for the GAS Source and run the environment script (`envas` or `envas.bat`) in your GAS installation directory.

4. Start the standalone dispatcher.

   `httpdispatch &`

5. Run the application in your browser.

   In this example, the URL is looking for a configuration file named `myapp.xcf`:

   `http://localhost:6394/ua/r/myapp`

   As you interact with the application, all this activity is recorded in the file specified.

6. To finish recording, close the application.

   **Warning:** Next time you run the application with this DVM configuration, the log file is overwritten.

When you have completed recording the log, your next task is to generate test sets from the recorded log file. See *Generate a scenario from a guilog* on page 39.

Record a GDC log

Before you begin:

The Genero Desktop Client (GDC) records actions from all running applications in its log. The GGC generates test scenarios from this log. Therefore, you need to ensure the log only records actions from your test application.
**Warning:** To avoid errors in the log caused by the GDC recording actions from other applications, it is highly recommended you close all applications running in the GDC before starting the recording for your test application.

For more information on GDC logging, see the *Using the logging system* page in the *Genero Desktop Client User Guide*.

**About this task:**

In this task you record a log from the GDC.

**Important:** The beginning (starting the application) and ending (stopping the application) must be included in the recorded log in order for the GGC to have a complete session. Therefore, you need to perform the recording steps in this order:

1. Start recording.
2. Launch the application you want to test.
3. Close the application before you stop the recording.

1. Launch the Genero Desktop Client.
2. Begin recording the GDC log file.
   - a) In the GDC monitor, select the **Debug** panel
   - b) In the **File** field, enter the path to the log file
      
      **Note:** If the file exists, it will be overwritten.
   - c) Click **Record**
3. Start the application using `fglrun`. For example:

```
D:\fjs\gst\fgl\demo\Widgets>fglrun Edit
```

4. Use the application to execute some actions to record in the log.

5. When you are finished, close the application.

6. To finish recording, click **Stop**

When you have completed recording the log, your next task is to generate test sets from the recorded log file. See *Generate a scenario from a GDC log* on page 39.

**Record a GBC log**

**About this task:**

The GGC can generate test scenarios from logs recorded by the Genero Browser Client. GBC version 1.00.53 or greater is required. In this task you record a log from the GBC. For more information, see the *Record a GBC log* page of the GBC documentation provided as part of the GBC project package.
1. Set the environment for the GAS.  
   Source and run the environment script (envas or envas.bat) in your GAS installation directory.

2. Start the standalone dispatcher.
   
   ```
   httpdispatch &
   ```

3. Run an application in your browser.  
   In this example, the "price" application is launched with the `recordGbcLog=1` string to set recording log mode. It records the session's interaction between the front-end and the DVM:

   ```
   http://localhost:6394/ua/r/price?recordGbcLog=1
   ```

   ![Figure 7: GBC log recording](https://via.placeholder.com/150)

   As you interact with the application, all this activity is recorded.

   **Note:** Notice the "Recording log ..." label in the application title bar.

4. To stop the recording, close the application:  
   When you close the application, the end page displays, with a link to "Get the GBC logs". Click on this link and follow the instructions in the system window that opens to download the session log.

   A GBC log file has a name like `price-e798a311.gbclog`. It takes the format of the `app_name` followed by a hyphen ("-"), part of the `session_id`, and the `gbclog` extension.

   The log is downloaded to your downloads directory.

When you have completed recording the log, your next task is to generate test sets from the recorded log file. See [Generate a scenario from a GBC log](#) on page 40.
Generate a scenario from a guilog

Generate test sets from recorded log files.

Before you begin:

Ensure your environment is configured to run Java and the Genero Ghost Client. See Install and configure for Genero Ghost Client on page 8.

About this task:

The `ggcgen` command transforms a DVM guilog into a BDL or a Java scenario. In this task you can choose the output you want by specifying it in the command.

Type one of the following commands to generate a scenario from the GUI log.

- BDL:
  ```
  ggcgen bdl myTest.guilog
  ```

- Java:
  ```
  ggcgen java myTest.guilog
  ```

In this example the `myTest.guilog` is the recorded log file.

The `ggcgen bdl` command creates the file `myTest.4gl` as the test scenario.

The `ggcgen java` command creates a scenario directory in your working directory where it saves a `myTest_provider.java` file.

When you have completed generating the test sets from the log, your next task is to compile and run them. See Write, compile, and run BDL tests on page 26 or Write, compile, and run Java tests on page 29.

Related tasks

Record a scenario log on page 34

Describes procedures used to record a log file of the interaction with an application launched to a Genero web client.

Generate a scenario from a GDC log

Generate test sets from recorded Genero Desktop Client log file.

Before you begin:

Ensure your environment is configured to run Java and the Genero Ghost Client. See Install and configure for Genero Ghost Client on page 8.

About this task:

The `ggcgen` command transforms a GDC log into a BDL or a Java scenario. In this task you can choose the output you want by specifying it in the command.

Type one of the following commands to generate a scenario from the GDC log.

- BDL:
  ```
  ggcgen bdl myGDCtest.log
  ```

- Java:
  ```
  ggcgen java myGDCtest.log
  ```

In this example the `myGDCtest.log` is the recorded GDC log file.

The `ggcgen bdl` command creates the scenario file `myGDCtest.4gl` in your working directory.
The `ggcgen java` command creates a scenario directory in your working directory where it saves the scenario file `myGDCTest_provider.java`.

When you have completed generating the test sets from the log, your next task is to compile and run them. See Write, compile, and run BDL tests on page 26 or Write, compile, and run Java tests on page 29.

**Related tasks**
- Record a scenario log on page 34
  Describes procedures used to record a log file of the interaction with an application launched to a Genero web client.

**Generate a scenario from a GBC log**

Generate test sets from recorded Genero Browser Client log file.

**Before you begin:**

Ensure your environment is configured to run Java and the Genero Ghost Client. See Install and configure for Genero Ghost Client on page 8.

**About this task:**

The `ggcgen` on page 55 command transforms a GBC log into a BDL or a Java scenario. In this task you can choose the output you want by specifying it in the command.

Type one of the following commands to generate a scenario from the GBC log.

- **BDL:**
  ```
ggcgen bdl myGBCtest.gbclog
  ```

- **Java:**
  ```
ggcgen java myGBCtest.gbclog
  ```

In this example the `myGBCtest.gbclog` is the recorded GBC log file.

The `ggcgen bdl` command creates the scenario file `myGBCtest.4gl` in your working directory.

The `ggcgen java` command creates a scenario directory in your working directory where it saves the scenario file `myGBCtest_provider.java`.

When you have completed generating the test sets from the log, your next task is to compile and run them. See Write, compile, and run BDL tests on page 26 or Write, compile, and run Java tests on page 29.

**Related tasks**
- Record a scenario log on page 34
  Describes procedures used to record a log file of the interaction with an application launched to a Genero web client.

**Set test environment**

There are options available to specify the environment for a test run in direct connection mode (tcp) to the Genero Desktop Client.

**About this task:**

The GGC has options to set environment for tests handled by `ggcadmin` on page 50 and BDL scenario program tcp commands.

By setting the environment option (`-e` or `--environment-file`), it allows you to get an environment variable, such as the database path (FGLDBPATH), of the tested program for the test execution.

In this task, you provide a list of environment variables in a text file.
Steps

1. Navigate to your test directory.
   This example uses a directory named /test_dir.
2. Create a text file in your test directory, myenv in the example.
3. Add entries for each environment variable you need to specify.
   For example:

   ```
   FGLGUIDEBUG=1
   FGLIMAGEPATH=${MYAPPROOT}$/images
   ${path.separator}${FGLDIR}/lib/image2font.txt"
   ```

   For OS independence:
   - Environment variables that need to be expanded are referenced using a dollar sign and curly braces (same for both UNIX® and Windows®).
   - The ${path.separator} predefined resource resolves a separator in list of paths to colon (:) on UNIX® platforms and semi-colon (;) on Windows®.
   - The ${file.separator} predefined resource resolves a separator in a file path to slash (/) on UNIX® platforms and back slash (\) on Windows®.
4. Type one of the following commands to run the test depending on whether your tests are written in BDL or Java.
   - BDL:
     ```
     fglrun myapp_test.42m tcp --environment-file myenv -c "fglrun myapp"
     ```
   - Java:
     ```
     ggcadmin tcp -w path/to/myapp --environment-file myenv -c "fglrun myapp"
     --scenario test_dir.myapp_test
     ```

   The test runs and the results display in the output.

Related concepts

**ggcadmin** on page 50
The ggcadmin is an administration tool providing commands to start and stop the BDL scenario server, and to run Java scenarios in either TCP direct or UA mode through the GAS.

**GGC scenario program options** on page 57
A generated GGC scenario implements a set of command line options.

---

Test speed ratio

There is an option available to specify the speed the generated test is played.

About this task:

The scenario can play at the pace specified in the generated scenario. There is a default or normal wait delay (defined as 1 or 100%) between each test instruction. By setting a speed ratio option, it allows you to speed up or slow down the pace of the test execution.

In this task, you set --speed-ratio to shorten the time it takes to complete the test.

Steps

1. Navigate to your test directory.
   This example uses a directory named /test_dir.
2. Type one of the following commands to run the test depending on whether your tests are written in BDL or Java.

   By setting `--speed-ratio` to 0.1, there is a 10% increase in speed. For an even faster test, set the ratio to 0, which equals no wait delay.

   - BDL:
     
     ```
     fglrun myapp_test.42m tcp --speed-ratio 0.1 -c "fglrun myapp"
     ```

   - Java:
     
     ```
     ggcadmin tcp -w path/to/myapp --speed-ratio 0.1 -c "fglrun myapp" --scenario test_dir.myapp_test
     ```

The test runs and the results display in the output.

**Tip:** Another option is to generate the test without wait delay instructions using the `--no-wait` option. For example:

- `ggcgen bdl --no-wait myapp.guilog`
  
  Instructions such as `CALL ggc.wait(...)` are then not generated in your BDL scenario.

- `ggcgen java --no-wait myapp.guilog`
  
  Instructions such as `client.wait(...)` are not generated in your Java scenario.

**Related concepts**

`ggcadmin` on page 50

The `ggcadmin` is an administration tool providing commands to start and stop the BDL scenario server, and to run Java scenarios in either TCP direct or UA mode through the GAS.

`ggcgen` on page 55

You use the `ggcgen bdl` tool to generate test scenarios from a recorded session. It accepts logs recorded by the DVM (guilog) or logs recorded using the Genero Desktop Client (GDC).

`GGC scenario program options` on page 57

A generated GGC scenario implements a set of command line options.

### Check certificates

The Genero Ghost Client by default does not enforce certificate checks.

The GGC provides you with an option to verify the server’s certificates against the recognized certificate authority. The `ggcadmin` on page 50 tool allows you to enable or disable certificates checks.

Although certificate checks provide more security, most of the time in a development environment you use self-signed certificates that are rejected. So in this case you would disable checks by default. This allows you to proceed and run your test even for server connections otherwise considered insecure.

When you want to enable certificate checks, include the `--check-certificates` option in the command to test the application, for example:

```
ggcadmin ua --check-certificates --url https://myserver:6394/ua/r/price --scenario demo.SimpleScenario
```

The server connection is verified by making sure the server's certificate contains the right name and verifies successfully using the cert store.
**Authenticate with single sign on**

The Genero Ghost Client allows you to test applications that require authentication through Single sign on. The GIP SSO Handler (GIPSSOHandler on page 111) provides support for SSO that can be used to authenticate using the Genero IDP SSO service.

The Java API `com.fourjs.ggc.HttpHandler` interface provides support for HTTP handling for SSO.

Specify the GIP SSO Handler with the option `--http-handler com.fourjs.ggc.httpHandler.GIPSSOHandler` with application SSO login details at the command line.

**Front calls**

As a developer, you must know how to handle front calls in your test.

The GGC has a set of implementations for standard front-end calls, such as those to get the front-end name, operating system type, etc., that are handled by default.

Default front calls provide a pre-defined answer. For example, if you use `standard.feInfo("feName")`, unless you want it to return "GGC", you will have to overload it and provide the values you expect it to return.

Overloading the defaults means you provide your own implementation of these functions, and you do not use the default implementation the GGC provides. You can implement overloaded functions in Genero BDL or Java.

Specific front call handlers are needed to Implement custom front calls on page 47. For sample code, see Example: custom front call (BDL) on page 47 or Example: custom front call (Java) on page 48.

The BDL API can also implement front call handlers through the Java API handler interface using option `--frontcall-handler frontcall_handler` at the command line. See Java front calls with BDL scenario on page 46. The front call handler option can be used for both BDL and Java.

If there is a front call in the test, the guilogs and the generated code (comments are generated in the generated code) show which front call is called, but it is not known when it is called. The scenario must therefore register a front call handler as a callback function. Otherwise the tested program may just stop simply because it does not receive the expected information, and an error like 'Not implemented' may be returned.

**Default front calls**

Front call implementations that are handled by default by the GGC.

Table 6: Default front calls on page 44 lists implemented front calls and the returned values.
Table 6: Default front calls

<table>
<thead>
<tr>
<th>Front call</th>
<th>Description</th>
<th>Returned value</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>standard.feInfo(&quot;feName&quot;)</code></td>
<td>Returns the front-end name, the Genero Ghost client is identified as GGC.</td>
<td>GGC</td>
</tr>
<tr>
<td><code>standard.feInfo(&quot;isActiveX&quot;)</code></td>
<td>Returns if the front-end runs in active X mode. The GGC does not run in active X mode.</td>
<td>0</td>
</tr>
<tr>
<td><code>standard.feInfo(&quot;osType&quot;)</code></td>
<td>Returns the operating system type where the front-end is running. Returns &quot;Windows&quot; if running on Windows, otherwise &quot;Unix&quot;.</td>
<td>&quot;Windows&quot; or &quot;Unix&quot;</td>
</tr>
<tr>
<td><code>standard.feInfo(&quot;numScreens&quot;)</code></td>
<td>Returns the number of screens available on the front-end platform.</td>
<td>1</td>
</tr>
<tr>
<td><code>standard.getEnv(&quot;NAME&quot;)</code></td>
<td>Returns the name of an environment variable, for example, <code>standard.getEnv(&quot;FGLDIR&quot;)</code></td>
<td>The value of the environment variable &quot;NAME&quot; on the front-end side.</td>
</tr>
</tbody>
</table>

For details of the Java packages and classes see the javadoc distributed with the Genero Ghost Client in your FGLGWS installation. The javadoc is found in your GGCDIR/doc/javadoc/ directory. See the help file by launching the /doc/javadoc/index.html file in your browser.

**Related concepts**

- [ggcadmin](#) on page 50
  The ggcadmin is an administration tool providing commands to start and stop the BDL scenario server, and to run Java scenarios in either TCP direct or UA mode through the GAS.

- [Implement custom front calls](#) on page 47
  Implement custom front calls with a custom front call handler.

**Register front call handler**

Register custom front calls with the front call handler.

**Note:** Default front calls do not need to be registered with a handler.

In your BDL tests, you can register front calls in two ways. Either:

- You register them in your BDL code in the `MAIN` block before the call to the `play` method with:

  ```
  call registerFrontCallHandler(fch FrontCallHandler)
  ```

  For an example, see [Implement front calls in a test](#) on page 44.

- Or you register them on the command line with the `--frontcall-handler` option. For an example, see [Java front calls with BDL scenario](#) on page 46.

In your Java tests, the front call is registered on the command line with the `--frontcall-handler` option.

**Implement front calls in a test**

Implementing a custom front call in your test.

In this task you implement a custom front call that overloads the default front call implementation to return the name of the front-end.

1. Write a function that takes a front call request as a parameter and returns the front call answer.
This function has the following format:

```plaintext
FUNCTION myCustomFrontCall (request ggc.FrontCallRequest INOUT) RETURNS ggc.FrontCallAnswer
```

See the sample function implementation of `standard.feInfo("feName")` in **Example: custom front call (BDL)** on page 47.

2. Register this function for callback before the function play is called.

For example, in the MAIN block of your scenario:

```plaintext
MAIN
  # Register custom front call handler
  CALL ggc.registerFrontCallHandler(FUNCTION myCustomFrontCall)
  # ...
  CALL play()
  EXIT PROGRAM 0
END MAIN
```

3. In your test application, make sure you have a front call action that you can test.

For example, a menu action:

```plaintext
MAIN
  DEFINE answer STRING
  OPEN FORM f FROM "frontcall"
  DISPLAY FORM f
  MENU "FrontCall"
    ON ACTION fename
      CALL ui.Interface.frontcall("standard","feinfo", ["fename"], [answer])
      MESSAGE answer
    END ACTION
    ON ACTION quit
      EXIT MENU
    END ACTION
END MENU
END MAIN
```

4. In the play function of your scenario test the action (fename in the example).

For example:

```plaintext
FUNCTION play()
  DEFINE msg ggc.Message
  CALL ggc.action("fename")
  CALL ggc.getMessage() RETURNING msg.*
  DISPLAY msg.message
  ...
END FUNCTION
```

**Related concepts**

- [Front call request](#) on page 102
- The Genero BDL API for GGC provides methods for front call requests.
- [Front call answer](#) on page 105
The Genero BDL API for GGC provides methods for answers to front call requests.

**Java front call parameters**

Java front calls can be configured with a parameter string.

To provide a parameterized front call function, the `FrontCallHandler setParameter(String parameter)` and `getParameter()` methods need to be overloaded:

```java
/**
 * Configure the HttpHandler parameters.
 * @param parameter The HTTP handler parameter string.
 * @return this - The current instance.
 * Default implementation ignores the parameter.
 */
FrontCallHandler setParameter(String parameter);

/**
 * Returns the HTTP Handler parameter
 * @return The parameter value, null if none;
 */
String getParameter();
```

Several common parameterized front calls are provided ready to use. These include:

- `standard.feInfo("feName")`
- `standard.openFile(...)`

For an example of the `OpenFile` function implementation see the `FGLDIR/testing_utilities/ggc/src/samples/functioncall` directory.

**Java front calls with BDL scenario**

BDL scenario can use Java implemented front calls.

You can use Java front call handlers in your BDL scenario, by specifying the option `--frontcall-handler` in the command line.

Multiple front call handlers `--frontcall-handler` can be registered in this way. If a front call handler does not process the front call request, the next registered handler is called.

The following sample command sends a request to use the Java `OpenFile` front call. This front call is implemented by the class `com.fourjs.ggc.frontcall.standard.OpenFile`. It allows you to provide a list of file paths that are returned in sequence.

The path to two files is specified; `/tmp/a.jpg` and `/tmp/b.jpg`, as an example. (Line breaks have been added to the command example to improve readability).

```
fglrun scenario ua
   --url http://localhost:6394/ua/r/app
```

For Java tests, the `ggcadmin on page 50` `ua` and `tcp` commands have the option to use the `--frontcall-handler also to request to use a front call handler.`

**Related reference**

Default front calls on page 43
Front call implementations that are handled by default by the GGC.

**Implement custom front calls**

Implement front calls with a custom front call handler.

You may need to overload and provide your own implementation for most of the default front calls. These are some examples of those you may need to overload:

- The default implementation of `standard.getEnv("NAME")` front call, will return the value of an environment variable, but this will apply to the GGC, for instance the BDL scenario server, if this was started. You will need to overload it to return the environment of the client.

- `standard.feInfo("numScreens")` returns the number of screens the client is working with. By default 1 is returned, but clients may have 2 or 3 screens. Therefore, you may need to implement your own call and provide your own values to the front call.

**Note:** There are two internal BDL front calls provided for `getFile` and `putFile` by default. It is not expected that these will need to be overloaded.

To provide a custom front call implementation, you must:

1. **Implement a function of the `FrontCallHandler` prototype:**

   ```plaintext
   PUBLIC TYPE FrontCallHandler FUNCTION(request FrontCallRequest INOUT) RETURNS FrontCallAnswer
   ```

   Where:
   
   - `FrontCallHandler` is the name of your custom function.
   - The parameter `request` of type `ggc.FrontCallRequest` is passed as a reference to the request information.
   - Values are returned in a record of type `ggc.FrontCallAnswer`.

   **Note:** For more information see the functions and record types defined in the `ggc.4gl` module source at `FGLDIR/testing_utilities/ggc/src/ggclib`.

2. **Register** the function for callback with the call handler function.

**Example: custom front call (BDL)**

Overloading the default `standard.feInfo("feName")` front call.

**Front call implementation for `standard.feInfo("feName")`**

In this function, a call to the `standard.feInfo("feName")` is implemented and other front calls are left unprocessed (`notProcessed()` on page 107). If an error is encountered, the front call chain stops and the error is returned to the DVM. Several errors can be returned in the answer object such as:

- `ggc.FrontCallAnswer.moduleNotFound()`
- `ggc.FrontCallAnswer.functionNotFound()`
- `ggc.FrontCallAnswer.stackError()`
- `ggc.FrontCallAnswer.userError(errorMessage STRING)`

```plaintext
FUNCTION customFrontCall(request ggc.FrontCallRequest INOUT) RETURNS ggc.FrontCallAnswer
    DEFINE answer ggc.FrontCallAnswer
    DISPLAY "MODULE: ", request.getModuleName()
    DISPLAY "FUNCTION: ", request.getFunctionName()
    DISPLAY "Parameter: ", request.getParameterValue(1)
    CALL answer.notProcessed()
    IF request.getModuleName() == "standard" AND request.getFunctionName() == "feinfo" THEN
        IF request.getParameterCount() == 0 THEN
```
CALL answer.stackError()
ELSE
  IF request.getParameterValue(1) == "fname" THEN
    IF request.getParameterCount() != 1 THEN
      CALL answer.stackError()
    ELSE
      CALL answer.success()
      CALL answer.returnString("MYGBC")
    END IF
  END IF
END IF
RETURN answer.*
END FUNCTION

Related concepts
Front call request on page 102
The Genero BDL API for GGC provides methods for front call requests.

Front call answer on page 105
The Genero BDL API for GGC provides methods for answers to front call requests.

Related reference
Default front calls on page 43
Front call implementations that are handled by default by the GGC.

Example: custom front call (Java)
Overloading the default standard.feInfo("feName") front call.

The Java API allows you to implement custom front calls for overloading existing default front call implementations, such as standard.feInfo("feName").

The Scenario class implements the FrontCallHandler interface providing a no-operation front call handler. To provide a custom implementation, either overload the getFrontCallHandler() method:

```java
@override
public FrontCallHandler getFrontCallHandler()
{
    return new MyFrontCallHandler(...);
}
```

Or overload the invoke() method:

```java
@override
public FrontCallEvent invoke(FrontCall frontCall)
```

Front call implementation for standard.feInfo("feName")
In this function, it shows how the default standard.feInfo("feName") front call can be overloaded.

- The FrontCall parameter contains the front call request object for module name, function name, parameters, and expected number of return values. See details of the FrontCall class in the GGCDIR/doc/javadoc/ directory.
- A front call can remain unprocessed by returning a FrontCallEvent.notProcessed(). In this case, the front call will be provided to the next front call handler. The last front call handler is the default front call handler provided by the GGC.
- If an error is encountered, the front call chain stops and the error is returned to the DVM. Several errors can be returned like:
  - ggc.FrontCallAnswer.moduleNotFound()
  - ggc.FrontCallAnswer.functionNotFound()
• `ggc.FrontCallAnswer.stackError()`
• `ggc.FrontCallAnswer.userError(errorMessage STRING)`

• In case of success, the front call can return values using the `FrontCallEvent.Builder` class. For example:

```java
return new FrontCallEvent.Builder(frontCall).success().returnValue("hello").returnValue("world")
```

```java
@Override
public FrontCallEvent invoke(frontCall frontCall) {
    String moduleName = frontCall.moduleName();
    String functionName = frontCall.functionName();

    /* Ensure module name is "standard" */
    if (!moduleName.equals("standard")) {
        return FrontCallEvent.notProcessed();
    }

    /* Ensure function name is "feinfo" */
    if (!functionName.equals("feinfo")) {
        return FrontCallEvent.notProcessed();
    }

    // At least one parameter is required.
    List parameters = frontCall.getParameters();
    if (parameters.size() == 0) {
        return FrontCallEvent.stackError();
    }

    /* Only process "fename" and return "GGC-Custom" */
    switch (parameters.get(0).getValue()) {
    case "fename":
        return new FrontCallEvent.Builder(frontCall).success().returnValue("GGC-Custom").build();
        case "fename":
            return new FrontCallEvent.Builder(frontCall).success().returnValue("GGC-Custom").build();
        default:
            return FrontCallEvent.notProcessed();
    }
}
```

---

**Genero Ghost Client reference**

Reference topics for the Genero Ghost Client.

**Tools and Commands**

Command line tools provided for GGC.

All tools accept the `@file` argument which allows you to load arguments from a file. Options for a command that you will type over and over again can be put into a file instead. Then just call the command with the file as argument. Tools can also load options automatically from a default argument file. See Default argument file on page 59.
**ggcadmin**

The ggcadmin is an administration tool providing commands to start and stop the BDL scenario server, and to run Java scenarios in either TCP direct or UA mode through the GAS.

**Syntax**

```
$ ggcadmin {-V | -h | command [options]}
```

1. `command` is one of the commands as described in **ggcadmin commands** on page 50.
2. `options` are described in **Options** on page 50.

**ggcadmin commands**

The `ggcadmin` tool supports four commands:

- **ggcadmin tcp [options]**
  
The `tcp` command launches the Java scenario in direct TCP mode. Options are described in **Table 8: ggcadmin tcp command options** on page 50.

- **ggcadmin ua [options]**
  
The `ua` command launches the Java scenario in UA mode. Options are described in **Table 9: ggcadmin ua command options** on page 52.

- **ggcadmin startbdls server [options]**
  
The `startbdls server` command starts the BDL scenario server. Options are described in **Table 10: ggcadmin startbdls server command options** on page 54.

- **ggcadmin stopbdls server [options]**
  
The `stopbdls server` command stops the BDL scenario server. Options are described in **Table 11: ggcadmin stopbdls server command options** on page 54.

**Options**

**Table 7: ggcadmin version and help options**

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>-V or --version</td>
<td>Display GGC version information.</td>
</tr>
<tr>
<td>-h or --help</td>
<td>Displays help for the <code>ggcadmin</code> commands.</td>
</tr>
</tbody>
</table>

**Table 8: ggcadmin tcp command options**

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>-h or</td>
<td>Displays help for the <code>tcp</code> command.</td>
</tr>
<tr>
<td>--help</td>
<td></td>
</tr>
<tr>
<td>-V or</td>
<td>Prints version information</td>
</tr>
<tr>
<td>--version</td>
<td></td>
</tr>
<tr>
<td>Option</td>
<td>Description</td>
</tr>
<tr>
<td>--------</td>
<td>-------------</td>
</tr>
<tr>
<td><code>-w directory_name</code> or <code>--working-directory directory_name</code></td>
<td>Specify the directory of the test application. The default is the current working directory (.).</td>
</tr>
<tr>
<td><code>-c argument</code> or <code>--command-line argument</code></td>
<td>Specify the runtime command (fglrun). The argument is the program passed to the runtime to test. Enclose the command and argument in quotations. For example, &quot;fglrun price&quot;.</td>
</tr>
<tr>
<td><code>--dump-command</code></td>
<td>Expand the command line options that are loaded from a default argument file. See Default argument file on page 59.</td>
</tr>
<tr>
<td><code>-e argument</code> or <code>--environment-file argument</code></td>
<td>Sets environment variables that will prepare the correct environment to run the test scenario. The argument is a file listing the environment variables in the form: ENVIRONMENT_VARIABLE_NAME=&lt;value&gt; If environment variables in values are to be expanded, they must be referenced (regardless of the platform) using the dollar sign and curly braces, for example ${ENVIRONMENT_VARIABLE_NAME}. For examples, see Set test environment on page 40.</td>
</tr>
<tr>
<td><code>--fname client-name</code></td>
<td>Change the default client name used by the GGC. Default client names: • When using an HTTP connection (via the Genero Application Server (/ua/)), GGC uses &quot;GBC&quot;. • When using a direct connection (TCP/IP) or a local connection, GGC uses &quot;GDC&quot;. The client name is used when connection strings are exchanged, to identify the client type. It is also the value returned when using the feinfo.fname built-in front call.</td>
</tr>
<tr>
<td><code>-f argument</code> or <code>--forward-gui argument</code></td>
<td>Forward GUI to the specified FGLSERVER. The AUI tree received from the DVM can be forwarded to a running GDC with <code>--forward-gui localhost:0</code> to provide a visual feedback of the scenario execution.</td>
</tr>
<tr>
<td><code>-F argument [...],...</code> or <code>--frontcall-handler argument [...],...</code></td>
<td>Provides front call handler support for user-provided front call handlers.</td>
</tr>
<tr>
<td><code>--scenario argument [...],...</code></td>
<td>Specify test scenarios to be run. The argument option can take a list of scenarios separated by commas.</td>
</tr>
<tr>
<td>Option</td>
<td>Description</td>
</tr>
<tr>
<td>--------</td>
<td>-------------</td>
</tr>
<tr>
<td><code>--scenario-provider argument</code></td>
<td>The scenario provider implements scenarios for an application starting a child application, which may require multiple scenarios. It expects a class implementing the <code>com.fourjs.ggc.ScenarioProvider</code> interface. The provider provides a scenario instance when requested. See <a href="#">Explore the scenario provider</a> on page 113. <strong>Important:</strong> The <code>--scenario-provider</code> option can not be used with the <code>--scenario</code> option and vice versa.</td>
</tr>
<tr>
<td><code>--session-ending-delay argument</code></td>
<td>Define an idle delay before considering the session ended. The <code>argument</code> is the delay in milliseconds you specify. The default is 1000.</td>
</tr>
<tr>
<td><code>-s argument</code> or <code>--speed-ratio argument</code></td>
<td>Specify an execution speed for the test. The scenario generator injects wait delays. This parameter provides a way to run your tests faster or slower. For example, a ratio set to 1 equals 100%. This is the default or normal wait delay. A 10% increase in speed is achieved by setting the ratio to 0.1. For an even faster test, set the ratio to 0, which equals no wait delay. Setting the ratio to 2, increases the delay by a factor 2 (200%), the scenario will take at least twice as long to execute. See <a href="#">Test speed ratio</a> on page 41.</td>
</tr>
</tbody>
</table>

### Table 9: ggcadmin `ua` command options

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>-h</code> or <code>--help</code></td>
<td>Displays help for the <code>ua</code> command.</td>
</tr>
<tr>
<td><code>-V</code> or <code>--version</code></td>
<td>Prints version information</td>
</tr>
<tr>
<td><code>-u argument</code> or <code>--url argument</code></td>
<td>Specify the URL to start the application. For example, <code>http://localhost:6394/ua/r/gwc-demo</code>.</td>
</tr>
<tr>
<td><code>--check-certificates</code></td>
<td>Request to check SSL certificates. This feature is disabled by default (false). To enable set <code>--check-certificates</code>. See <a href="#">Check certificates</a> on page 42.</td>
</tr>
<tr>
<td><code>--dump-command</code></td>
<td>Expand the command line options that are loaded from a default argument file. See <a href="#">Default argument file</a> on page 59.</td>
</tr>
<tr>
<td>Option</td>
<td>Description</td>
</tr>
<tr>
<td>--------</td>
<td>-------------</td>
</tr>
</tbody>
</table>
| **--fname client-name** | Change the default client name used by the GGC.  
Default client names:  
- When using an HTTP connection (via the Genero Application Server(/ua/)), GGC uses "GBC".  
- When using a direct connection (TCP/IP) or a local connection, GGC uses "GDC".  
The client name is used when connection strings are exchanged, to identify the client type. It is also the value returned when using the `feinfo.fname` built-in front call. |
| **-f argument or --forward-gui argument** | Forward GUI to the specified FGLSERVER. The AUI tree received from the DVM can be forwarded to a running GDC with `--forward-gui localhost:0` to provide a visual feedback of the scenario execution. |
| **-F argument [,...] or --frontcall-handler argument [,...]** | Provides front call handler support for user-provided front call handlers. |
| **-H argument [,...] or --http-handler http_handler [,...]** | Provides HTTP handler support. For example, a default HTTP handler (`GIPSSOHandler` on page 111) is provided for Single Sign on to authenticate a user with the Genero IDP SSO. |
| **--scenario argument [,...]** | Specify test scenarios to be run. The `argument` option can take a list of scenarios separated by commas. |
| **--scenario-provider argument** | The scenario provider implements scenarios for an application starting a child application, which may require multiple scenarios.  
It expects a class implementing the `com.fourjs.ggc.ScenarioProvider` interface.  
The provider provides a scenario instance when requested. See `Explore the scenario provider` on page 113.  
**Important:** The `--scenario-provider` option cannot be used with the `--scenario` option and vice versa. |
| **--session-ending-delay argument** | Define an idle delay before considering the session ended. The `argument` is the delay in milliseconds you specify. The default is 1000. |
| **-s argument or --speed-ratio argument** | Specify an execution speed for the test. The scenario generator injects wait delays. This parameter provides a way to run your tests faster or slower. For example, a ratio set to 1 equals 100%. This is the default or normal wait delay. A 10% increase in speed is achieved by setting the ratio to 0.1. For an even faster test, set the ratio to 0, which equals no wait delay. Setting the ratio to 2, increases the delay by a factor 2 (200%), the scenario will take at least twice as long to execute. See `Test speed ratio` on page 41. |
Table 10: \texttt{ggcadmin startbdlserver} command options

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>-h or --help</td>
<td>Displays help for the \texttt{startbdlserver} command.</td>
</tr>
<tr>
<td>-V or --version</td>
<td>Prints version information</td>
</tr>
<tr>
<td>--dump-command</td>
<td>Expand the command line options that are loaded from a default argument file. See \texttt{Default argument file} on page 59.</td>
</tr>
<tr>
<td>-i \textit{argument} or --idle-delay \textit{argument}</td>
<td>Define an idle delay in seconds to wait before exiting due to inactivity. The \textit{argument} is the delay in seconds you specify. The default is 300. 0 is infinite wait.</td>
</tr>
<tr>
<td>-p \textit{argument} or --port \textit{argument}</td>
<td>Specifies the port on which the server listens for connections (default is 6500)</td>
</tr>
</tbody>
</table>

Table 11: \texttt{ggcadmin stopbdlserver} command options

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>-h or --help</td>
<td>Displays help for the \texttt{stopbdlserver} command.</td>
</tr>
<tr>
<td>-V or --version</td>
<td>Prints version information</td>
</tr>
<tr>
<td>--dump-command</td>
<td>Expand the command line options that are loaded from a default argument file. See \texttt{Default argument file} on page 59.</td>
</tr>
<tr>
<td>-p \textit{argument} or --port \textit{argument}</td>
<td>Specifies the port on which the server listens for connections (default is 6500)</td>
</tr>
</tbody>
</table>

Usage

The \texttt{ggcadmin} command line tool provides two main functions for working with scenarios:

\textbf{Start and stop the BDL scenario server}

The command \texttt{ggcadmin startbdlserver} can be used to start the scenario server manually. Generally, you don't need to use this command as the scenario implicitly starts the daemon of the BDL server, and therefore it stops when the scenario ends.

If you start the scenario server manually, it will keep running when the scenario ends and then you need to use the \texttt{ggcadmin stopbdlserver} command to stop it.

\textbf{Run Java scenario tests}

You use the \texttt{ggcadmin tcp} command to run Java tests in direct mode. For example:
You use the `ggcadmin ua` command to run Java tests against applications running in the Genero Application Server (GAS). The application must be open at the URL:

```
ggcadmin ua --url http://localhost:6394/ua/r/myApp --scenario myTests.MyScenario
```

See How to run Java tests on page 32.

**ggcgen**

You use the `ggcgen` bdl tool to generate test scenarios from a recorded session. It accepts logs recorded by the DVM (`guilog`) or logs recorded using the Genero Desktop Client (GDC).

**Syntax**

```
ggcgen [-V | -h | command [options] argument [...] ]
```

1. `command` is either `bdl` to generate a test scenario in BDL, or `java` to generate a scenario in Java.
2. `options` are described in Options on page 55.
3. `argument` are log files or skeleton names passed as argument from which scenarios are generated. One scenario per log file. With the `--skeleton` option, an empty scenario is generated without using a log file.

**Options**

**Table 12: ggcgen version, help, and output options**

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>-V or --version</td>
<td>Display GGC version information.</td>
</tr>
<tr>
<td>-h or --help</td>
<td>Displays help for the <code>ggcgen</code> commands.</td>
</tr>
<tr>
<td>-v or --verbose</td>
<td>Generate verbose output for help with debugging. This option is not enabled by default.</td>
</tr>
<tr>
<td>--output-directory</td>
<td>Specify where to place generated files. Default is current working directory.</td>
</tr>
<tr>
<td>--comment-prefix <code>argument</code></td>
<td>The template's comment prefix. The default punctuation is two exclamation marks, <code>!!</code>, and lines with this prefix are filtered. The <code>argument</code> option allows you to specify an alternative prefix.</td>
</tr>
<tr>
<td>--dump-command</td>
<td>Expand the command line options that are loaded from a default argument file. See Default argument file on page 59.</td>
</tr>
</tbody>
</table>

**Table 13: ggcgen template directory option**

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>--template-directory <code>argument</code> [ ... ]</td>
<td>Specify template directories to use. The <code>argument</code> option can take a list of directories. The lookup will be performed in the order the arguments appear in the command line.</td>
</tr>
</tbody>
</table>
### Table 14: ggcgen dump/wait options

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>--no-wait</td>
<td>Specify that wait instructions, such as BDL Call ggc.wait() or Java client.wait(), are not generated. The 'wait' instruction is expanded using the wait.4gl code template.</td>
</tr>
<tr>
<td>--dump-all</td>
<td>Enables all information (AUI log, AUI tree) to be included in comments in the scenario. Default value is false.</td>
</tr>
<tr>
<td>--dump-auitree</td>
<td>The state of the AUI tree is included in comments between each instruction sent to the DVM. Default value is false.</td>
</tr>
<tr>
<td>--dump-auilog</td>
<td>The source log file used to generate the scenario is included in comments in the instructions between the AUI tree and the client. Default value is false.</td>
</tr>
</tbody>
</table>

### Table 15: ggcgen check options

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>--check-all</td>
<td>This option provides you with optional checks that can be generated and then performed at runtime. It enables all checks provided by the templates in GGCDIR/template/bdl/check_*.*4gl.</td>
</tr>
<tr>
<td>--check-window</td>
<td>This option checks the current window name and title using the template GGCDIR/template/bdl/check_window.4gl</td>
</tr>
<tr>
<td>--check-form</td>
<td>This option checks the current form name and title using the template GGCDIR/template/bdl/check_form.4gl</td>
</tr>
<tr>
<td>--check-focus</td>
<td>This option checks the focused field using the template GGCDIR/template/bdl/check_focus.4gl</td>
</tr>
<tr>
<td>--check-value</td>
<td>This option checks the value of the focused field using the template GGCDIR/template/bdl/check_value.4gl</td>
</tr>
<tr>
<td>--check-actions</td>
<td>This option checks the active/inactive state of an action using the template GGCDIR/template/bdl/check_action.4gl</td>
</tr>
<tr>
<td>--check-messages</td>
<td>This option generates code to check that MESSAGE and ERROR instructions contain expected messages values. It also checks in case values are missing. The GGCDIR/template path has the following templates that enable these checks based on whether your test uses Genero BDL or Java:</td>
</tr>
<tr>
<td></td>
<td>• check_message</td>
</tr>
<tr>
<td></td>
<td>• check_no_message</td>
</tr>
<tr>
<td></td>
<td>• check_error</td>
</tr>
<tr>
<td></td>
<td>• check_no_error</td>
</tr>
</tbody>
</table>

### Table 16: ggcgen skeleton option

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>--skeleton</td>
<td>This option allows you to generate an empty scenario. The header.4gl and footer.4gl templates only will be generated in the named scenario. Dump and check options will be ignored.</td>
</tr>
<tr>
<td>Option</td>
<td>Description</td>
</tr>
<tr>
<td>------------------------</td>
<td>------------------------------------------------------------------------------</td>
</tr>
<tr>
<td><code>--package-name</code></td>
<td>This option is used with the <code>ggcgen java --skeleton</code> command. It allows you to specify the Java package name.</td>
</tr>
</tbody>
</table>

**Usage**

You use the `ggcgen` tool to generate scenarios from a recorded session.

There are several options you can use to effect the generated scenario. You can set the template directory to one with your specific checks. Or if you want to perform standard checks, you can set these with the check options.

For example, for BDL:

```
ggcgen bdl --template-directory ${GGCDIR}/template/bdl/alternate-checks --check-all mylog.log
```

Or for Java:

```
ggcgen java --template-directory ${GGCDIR}/template/bdl/alternate-checks --check-all mylog.log
```

There are options to debug (`--dump-*`) when you need to see detailed information such as the state of the AUI tree generated in the scenario.

**Note:** In the normal course of testing it is not expected that you will need to use the dump options, except perhaps if the scenario does not behave as expected. Then generating more verbose details may help you to understand where and why it fails.

You can use the `--skeleton` option to generate an empty scenario. The code from the header and footer templates only will be expanded. It does not require a log file, just the name of the scenario to generate. Multiple scenarios can be created by listing them separated by spaces.

For example, for BDL:

```
ggcgen bdl --skeleton tests
```

Or for Java:

```
ggcgen java --skeleton --package-name tests myscenario
```

**GGC scenario program options**

A generated GGC scenario implements a set of command line options.

By default, a GGC scenario program generated by `ggcgen` implements a set of options to control the testing parameters.

Use the following command to display available scenario program options:

```
fglrun ggc-scenario-program \ua \tcp \--help
```

For example:

```
fglrun price_sample.42m \ua \--help
```

Information is displayed in the standard output for available options for BDL tests via the GAS (\ua) and in direct mode (\tcp).

**Note:** For Java tests, see the `ggcadmin` on page 50 command options.

The BDL options are described in the following tables:
**Table 17: Common GGC scenario program options**

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>-h or --help</td>
<td>Displays help for the <code>ua</code> and <code>tcp</code> commands.</td>
</tr>
<tr>
<td>-p argument or --port</td>
<td>Specifies the port on which the server listens for connections (default is 6500)</td>
</tr>
<tr>
<td>-s argument or --speed-ratio</td>
<td>Specify an execution speed for the test. The scenario generator injects wait delays. This parameter provides a way to run your tests faster or slower. For example, a ratio set to 1 equals 100%. This is the default or normal wait delay. A 10% increase in speed is achieved by setting the ratio to 0.1. For an even faster test, set the ratio to 0, which equals no wait delay. Setting the ratio to 2, increases the delay by a factor 2 (200%), the scenario will take at least twice as long to execute. See Test speed ratio on page 41.</td>
</tr>
<tr>
<td>-f argument or --forward-gui</td>
<td>Forward GUI to the specified FGLSERVER. The AUI tree received from the DVM can be forwarded to a running GDC with <code>--forward-gui localhost:0</code> to provide a visual feedback of the scenario execution.</td>
</tr>
<tr>
<td>--dump-command</td>
<td>Expand the command line options that are loaded from a default argument file. See Default argument file on page 59.</td>
</tr>
<tr>
<td>--fname client-name</td>
<td>Change the default client name used by the GGC. Default client names:</td>
</tr>
<tr>
<td></td>
<td>• When using an HTTP connection (via the Genero Application Server (<code>/ua/</code>)), GGC uses &quot;GBC&quot;.</td>
</tr>
<tr>
<td></td>
<td>• When using a direct connection (TCP/IP) or a local connection, GGC uses &quot;GDC&quot;. The client name is used when connection strings are exchanged, to identify the client type. It is also the value returned when using the <code>feinfo.fname</code> built-in front call.</td>
</tr>
</tbody>
</table>

**Table 18: GGC scenario program options specific to the `ua` command**

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>-H argument [,...] or --http-handler http_handler [,...]</td>
<td>Provides HTTP handler support. For example, a default HTTP handler (<code>GIPSSOHandler</code> on page 111) is provided for Single Sign on to authenticate a user with the Genero IDP SSO.</td>
</tr>
<tr>
<td>-F argument [,...] or --frontcall-handler argument [,...]</td>
<td>Provides front call handler support for user- provided front call handlers.</td>
</tr>
</tbody>
</table>
Table 19: GGC scenario program options specific to the `tcp` command

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>-w directory_name</code> or <code>--working-directory directory_name</code></td>
<td>Specify the directory of the test application. The default is the current working directory (<code>.</code>).</td>
</tr>
<tr>
<td><code>-F argument [...],...</code> or <code>--frontcall-handler argument [...],...</code></td>
<td>Provides front call handler support for user-provided front call handlers.</td>
</tr>
</tbody>
</table>
| `-e argument` or `--environment-file argument` | Sets environment variables that will prepare the correct environment to run the test scenario. The `argument` is a file listing the environment variables in the form:

```
ENVIRONMENT_VARIABLE_NAME=<value>
```

If environment variables in values are to be expanded, they must be referenced (regardless of the platform) using the dollar sign and curly braces, for example `${ENVIRONMENT_VARIABLE_NAME}`. For examples, see Set test environment on page 40. |
| `-c argument` or `--command-line argument` | Specify the runtime command (`fglrun`). The `argument` is the program passed to the runtime to test. Enclose the command and argument in quotations. For example, "fglrun price". |

Refer to the *Genero Business Development Language User Guide* for details regarding the `fglrun` command.

**Default argument file**

GGC tools can load command options from a default argument file named in a specific way in default locations.

**Syntax**

```
\_ tool.command \_ .tool.command \
```

1. `tool` is the name of the GGC tool. For example, `ggcadmin` on page 50, or `ggcgen` on page 55. For BDL scenario, the tool is "bdl".
2. `command` is the name of the tool command. For example, `startbdlservers`.

The file name syntax option with a dot (.) prefix is used to indicate a hidden file.

**Usage**

`ggcadmin`, `ggcgen`, and BDL scenario tools accept arguments from a file. This allows you to save options to a file that tend to be long and that otherwise you would type over and over again.
For example, you can save the options to start the bdl server in a file named `ggcadmin.startbdlserver` in your user home path `.ggc` directory.

```
--idle-delay=2000
--port=9000
```

When you run the associated command, the options are automatically loaded from the file. However, the GGC expects to find default files in either your ${HOME}/.ggc/ (Linux®) or %HOMEPATH%/.ggc/ (Windows®) directory, or in your current working directory.

**Note:** It uses the first file it finds.

**Table 20: Commands and default argument files**

<table>
<thead>
<tr>
<th>Command type</th>
<th>File name</th>
<th>Example</th>
</tr>
</thead>
</table>
| `ggcadmin command` | `ggcadmin.command` | `.ggcadmin.startbdlserver`
| BDL scenarios (for example, `fglrun scenario.42m tcp`) | `bdl.command` | `.bdl.tcp`
| `ggcgen command` | `ggcgen.command` | `.ggcgen.bdl`

**bdl tcp or ua**

For BDL scenarios run with `tcp` type command (for example, `fglrun scenario.42m tcp`), the `ggc` will look for the argument file in one of these formats:

```
.bdl.tcp
bdl.tcp
```

For BDL scenarios run with the `ua` type command (for example, `fglrun scenario.42m ua --url`), the `ggc` expects the argument file in one of these formats:

```
.bdl.ua
bdl.ua
```

**Show the expanded command line**

If you wish to see what options are loaded with the command, use the `--dump-command` option.

```
ggcadmin startbdlserver --dump-command
```

The output is shown:

```
  ggcadmin startbdlserver [--idle-delay=2000] [--port=9000]
```

**Display GGC version number**

To display the version of the Genero Ghost Client, execute `fpi -l` from the command line.
Refer to the *Genero Business Development Language User Guide* for details regarding the `fpi` command.

## The logging mechanism

The Genero Ghost Client implements a logging mechanism to display errors, warnings, and information as output from the tests. Learn how to control the output, and use the logging mechanism efficiently to detect and solve application problems.

### log.properties

The `log.properties` file defines the logging level for the Genero Ghost Client. It allows you to control the output messages.

Setting log properties define the level of detail and type of information you want outputted as tests are run. Locate the `log.properties` file in your Windows® `%HOMEPATH%/.ggc/` or Linux® `$HOME/.ggc/` directory.

### Sample log.properties file

```properties
# Log configuration
# Thu Apr 05 11:31:27 CEST 2018
# console properties
console.enabled=true
console.columns=relative-time contexts event-type event-params
console.categories=ERROR WARNING INFO VM HTTP ALL DEBUG
console.maxlength=4096
console.format=TEXT
# file properties
file.maxlength=-1
file.path=/tmp/ggc
file.enabled=true
file.columns=date time relative-time contexts event-type event-params
file.format=TEXT
file.categories=ERROR WARNING INFO VM HTTP DEBUG
```

You can set log properties for output to the console or a log file using the `console.enabled` and `file.enabled` properties and specifying category and column properties for each.

### Category property

The `.categories` properties sets the type of data written to the output. Most of the time, you will find that `ERROR` and `WARNING` logs are sufficient. Detailed log files may be requested to be sent to Four Js Support for analysis.

If more than one category is specified, they are separated by spaces.

```properties
file.categories=ERROR WARNING INFO VM HTTP
```

Valid data categories are shown in Table 21: Log categories on page 61.

### Table 21: Log categories

<table>
<thead>
<tr>
<th>Category</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ERROR</td>
<td>Logs errors. These are mostly fatal errors, like VM failing to start.</td>
</tr>
<tr>
<td>WARNING</td>
<td>Logs warnings.</td>
</tr>
<tr>
<td>INFO</td>
<td>Logs informational messages.</td>
</tr>
<tr>
<td>Category</td>
<td>Description</td>
</tr>
<tr>
<td>----------</td>
<td>-------------</td>
</tr>
<tr>
<td>VM</td>
<td>Logs all data exchanged with the DVM, data sent and received.</td>
</tr>
<tr>
<td>HTTP</td>
<td>Logs some HTTP level information - actual content should be reviewed.</td>
</tr>
<tr>
<td>DEBUG</td>
<td>Logs debug messages only. Expect this category to be highly verbose. <strong>Important:</strong> It is not advised to set DEBUG unless requested by support.</td>
</tr>
<tr>
<td>ALL</td>
<td>Logs all enabled categories, for example, errors, warnings, info, debug, etc.</td>
</tr>
</tbody>
</table>

### Column property

The `.column` property sets the fields that are displayed in the output. If more than one field is specified, they are separated by spaces. The order of the fields determines the order of the log output.

```
file.columns=date time relative-time contexts event-type event-params
```

Valid field identifier types are shown in Table 22: Field identifiers on page 62.

**Table 22: Field identifiers**

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>date</td>
<td>Date of logging the event.</td>
</tr>
<tr>
<td>time</td>
<td>Time of logging the event.</td>
</tr>
<tr>
<td>relative-time</td>
<td>Time elapsed since the test started.</td>
</tr>
<tr>
<td>contexts</td>
<td>Internal data representing the successive context the process went through to reach the logged event, for example, which VM, which HTTP request, etc.</td>
</tr>
<tr>
<td>event-type</td>
<td>Type of event logged.</td>
</tr>
<tr>
<td>event-params</td>
<td>Event details or message to be logged. Typically, just a one line log message.</td>
</tr>
</tbody>
</table>

### Log files

You specify the directory log files are created in by setting the `file.path` property. The default location, if not set, is the `%HOMEPATH%/.ggc/current_date` directory in a Windows® platform or `$HOME/.ggc/current_date` directory in a Linux® type platform. It is recommended to set the location to a temp directory on your system to avoid an unnecessary build up of these log files.

**Related concepts**

*Log files* on page 62

When you run a test, the Ghost Client creates log files which may be viewed for troubleshooting.

**Log files**

When you run a test, the Ghost Client creates log files which may be viewed for troubleshooting.

The following log files are created:

- `tcp-<session-id>.log` A log file is generated for the GDC direct tests started.
Log files have similar formats. They contain a list of the field identifiers that define the detail in the logged output. This appears at the start of the file:

```
#Fields: date time relative-time location thread-id contexts event-type event-params
```

As the application runs, messages are written for HTTP requests and responses, TCP or ua sessions started, system messages, etc. This output depends on the categories of logs specified in the `log.properties` on page 61 configuration.

**Accessing log files**

You can access log files in your Windows® `%HOMEPATH%/ggc/current_date` or Linux® `$HOME/.ggc/current_date` directory.

---

**Genero BDL API for GGC**

The Genero BDL API for the GGC provides types, functions, and methods for generating or writing tests in Genero BDL.

**Import the API**

The `ggc.42m` module in the `GGCDIR/lib/` directory defines the API. To implement it in your test, you need to add the instruction `IMPORT FGL ggc` to your Genero BDL test module.
<table>
<thead>
<tr>
<th>Types</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>PUBLIC TYPE <code>FrontCallAnswer</code> on page 75 RECORD</td>
<td>The <code>FrontCallAnswer</code> type defines a record for retrieving the result of a front call.</td>
</tr>
<tr>
<td>type STRING,</td>
<td></td>
</tr>
<tr>
<td>status STRING,</td>
<td></td>
</tr>
<tr>
<td>errorMessage STRING,</td>
<td></td>
</tr>
<tr>
<td>returnValues DYNAMIC ARRAY</td>
<td></td>
</tr>
<tr>
<td>OF RECORD isNull BOOLEAN,</td>
<td></td>
</tr>
<tr>
<td>value STRING,</td>
<td></td>
</tr>
<tr>
<td>index INTEGER,</td>
<td></td>
</tr>
<tr>
<td>dataType STRING END RECORD</td>
<td></td>
</tr>
<tr>
<td>END RECORD END RECORD</td>
<td></td>
</tr>
<tr>
<td>PUBLIC TYPE <code>FrontCallRequest</code> on page 75 RECORD</td>
<td>The <code>FrontCallRequest</code> type defines a record for retrieving front call request details.</td>
</tr>
<tr>
<td>frontCall RECORD moduleName STRING,</td>
<td></td>
</tr>
<tr>
<td>functionName STRING,</td>
<td></td>
</tr>
<tr>
<td>paramCount INTEGER,</td>
<td></td>
</tr>
<tr>
<td>returnCount INTEGER,</td>
<td></td>
</tr>
<tr>
<td>parameters DYNAMIC ARRAY</td>
<td></td>
</tr>
<tr>
<td>OF RECORD dataType STRING,</td>
<td></td>
</tr>
<tr>
<td>isNull BOOLEAN,</td>
<td></td>
</tr>
<tr>
<td>value STRING,</td>
<td></td>
</tr>
<tr>
<td>END RECORD END RECORD,</td>
<td></td>
</tr>
<tr>
<td>errorMessage STRING</td>
<td></td>
</tr>
<tr>
<td>END RECORD</td>
<td></td>
</tr>
<tr>
<td>PUBLIC TYPE <code>Message</code> on page 76 RECORD</td>
<td>The <code>Message</code> type defines a record for retrieving errors and messages.</td>
</tr>
<tr>
<td>isErrorMessage INTEGER,</td>
<td></td>
</tr>
<tr>
<td>message STRING</td>
<td></td>
</tr>
<tr>
<td>END RECORD</td>
<td></td>
</tr>
<tr>
<td>PUBLIC TYPE <code>Statistics</code> on page 76 RECORD</td>
<td>The <code>Statistics</code> type defines a record for retrieving test result statistics.</td>
</tr>
<tr>
<td>sessionId STRING,</td>
<td></td>
</tr>
<tr>
<td>bytesSent STRING,</td>
<td></td>
</tr>
<tr>
<td>bytesReceived INTEGER,</td>
<td></td>
</tr>
<tr>
<td>scenarioCount INTEGER,</td>
<td></td>
</tr>
<tr>
<td>scenarioFailed INTEGER,</td>
<td></td>
</tr>
<tr>
<td>downloadCount INTEGER,</td>
<td></td>
</tr>
<tr>
<td>downloadFailed INTEGER,</td>
<td></td>
</tr>
<tr>
<td>downloadBytes INTEGER,</td>
<td></td>
</tr>
<tr>
<td>sessionDuration RECORD start</td>
<td></td>
</tr>
<tr>
<td>Time BIGINT,</td>
<td></td>
</tr>
<tr>
<td>endTime BIGINT</td>
<td></td>
</tr>
<tr>
<td>END RECORD, downloadFailures DYNAMIC ARRAY OF STRING,</td>
<td></td>
</tr>
<tr>
<td>errors DYNAMIC ARRAY OF</td>
<td></td>
</tr>
<tr>
<td>RECORD status STRING,</td>
<td></td>
</tr>
<tr>
<td>detailMessage STRING</td>
<td></td>
</tr>
<tr>
<td>END RECORD, checkFailures DYNAMIC ARRAY OF RECORD</td>
<td></td>
</tr>
<tr>
<td>status STRING,</td>
<td></td>
</tr>
<tr>
<td>detailMessage STRING</td>
<td></td>
</tr>
<tr>
<td>END RECORD, checkFailures DYNAMIC</td>
<td></td>
</tr>
</tbody>
</table>
### Table 24: Functions testing actions and keys

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>action(name STRING)</code></td>
<td>Execute an action by name.</td>
</tr>
<tr>
<td><code>key(name STRING)</code></td>
<td>Send a key by name.</td>
</tr>
</tbody>
</table>

### Table 25: Functions for interacting with simple form fields

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>setFieldValue(fieldName STRING, value STRING)</code></td>
<td>Set value in a form field.</td>
</tr>
<tr>
<td><code>setFocus(fieldName STRING)</code></td>
<td>Set focus on a field.</td>
</tr>
<tr>
<td><code>setValue(value STRING)</code></td>
<td>Set value in the current field.</td>
</tr>
</tbody>
</table>
### Table 26: Functions for interacting with tables

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>getColumnValue</code></td>
<td>Get the column value of a table, tree, or screen record at the specified row.</td>
</tr>
<tr>
<td><code>getColumnValues</code></td>
<td>Get column values of a table, tree, or screen record.</td>
</tr>
<tr>
<td><code>hideTableColumn</code></td>
<td>Hide a table column.</td>
</tr>
<tr>
<td><code>setCellFocus</code></td>
<td>Select a cell in a table.</td>
</tr>
<tr>
<td><code>setRowFocus</code></td>
<td>Select a table row.</td>
</tr>
<tr>
<td><code>setRowSelection</code></td>
<td>Update a multiple row selection state.</td>
</tr>
<tr>
<td><code>setTableOffset</code></td>
<td>Set the table page size offset value.</td>
</tr>
<tr>
<td><code>setTableSize</code></td>
<td>Set the table size.</td>
</tr>
<tr>
<td><code>showTableColumn</code></td>
<td>Show a hidden table column.</td>
</tr>
<tr>
<td><code>sortTable</code></td>
<td>Sort a table.</td>
</tr>
</tbody>
</table>
Table 27: Functions for interacting with tree views

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>collapseTree</code></td>
<td>Collapse a node (row) in a tree view.</td>
</tr>
<tr>
<td><code>expandTree</code></td>
<td>Expand a node (row) in a tree view.</td>
</tr>
</tbody>
</table>

```sql
collapseTree (treeName STRING, row INTEGER )
expandTree (treeName STRING, row INTEGER )
```
<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>getFieldValue (fieldName STRING)</code> RETURNS STRING</td>
<td>Return the value in a specified form field.</td>
</tr>
<tr>
<td><code>getFieldValues (name STRING)</code> RETURNS DYNAMIC ARRAY OF STRING</td>
<td>Return a list of values from a table, tree, or screen record.</td>
</tr>
<tr>
<td><code>getFocus ()</code> RETURNS STRING</td>
<td>Return the name of the current focused element.</td>
</tr>
<tr>
<td><code>getFormName ()</code> RETURNS STRING</td>
<td>Get the current form name.</td>
</tr>
<tr>
<td><code>getFormTitle ()</code> RETURNS STRING</td>
<td>Get the current form title.</td>
</tr>
<tr>
<td><code>getValue ()</code> RETURNS STRING</td>
<td>Return the value in the current field.</td>
</tr>
<tr>
<td><code>getValues ()</code> RETURNS DYNAMIC ARRAY OF STRING</td>
<td>Returns the values of the current row of the current table, tree, or matrix.</td>
</tr>
<tr>
<td><code>getWidgetType (fieldName STRING)</code> RETURNS STRING</td>
<td>Return the widget type of the specified field.</td>
</tr>
<tr>
<td><code>getWindowName ()</code> RETURNS STRING</td>
<td>Get the current window name.</td>
</tr>
<tr>
<td><code>getWindowTitle ()</code> RETURNS STRING</td>
<td>Get the current window title.</td>
</tr>
</tbody>
</table>
Table 29: Functions for retrieving messages and errors

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>getError()</code></td>
<td>Return the current error message.</td>
</tr>
<tr>
<td></td>
<td><code>RETURNS ggc.Message</code></td>
</tr>
<tr>
<td><code>getMessage()</code></td>
<td>Return the current message.</td>
</tr>
<tr>
<td></td>
<td><code>RETURNS ggc.Message</code></td>
</tr>
</tbody>
</table>

Table 30: Functions for retrieving action state

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>isActionActive(name STRING)</code></td>
<td>Return the state of the specified action.</td>
</tr>
<tr>
<td></td>
<td><code>RETURNS BOOLEAN</code></td>
</tr>
</tbody>
</table>

Table 31: Functions for retrieving information about tables

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>getColumnCount(tableName STRING)</code></td>
<td>Get the number of columns in a table.</td>
</tr>
<tr>
<td></td>
<td><code>RETURNS INTEGER</code></td>
</tr>
<tr>
<td><code>getColumnName(tableName STRING, idx INTEGER)</code></td>
<td>Get a table column name.</td>
</tr>
<tr>
<td></td>
<td><code>RETURNS STRING</code></td>
</tr>
<tr>
<td><code>getCurrentColumn(tableName STRING)</code></td>
<td>Get the current column in a table.</td>
</tr>
<tr>
<td></td>
<td><code>RETURNS INTEGER</code></td>
</tr>
<tr>
<td><code>getCurrentRow(tableName STRING)</code></td>
<td>Get the current row of the table.</td>
</tr>
<tr>
<td></td>
<td><code>RETURNS INTEGER</code></td>
</tr>
<tr>
<td><code>getTableOffset(tableName STRING)</code></td>
<td>Get the table's current offset value.</td>
</tr>
<tr>
<td></td>
<td><code>RETURNS INTEGER</code></td>
</tr>
<tr>
<td><code>getTableSize(tableName STRING)</code></td>
<td>Get the table size.</td>
</tr>
<tr>
<td></td>
<td><code>RETURNS INTEGER</code></td>
</tr>
</tbody>
</table>
### Table 32: Functions for retrieving session information

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>getApplicationName()</code></td>
<td>Retrieve the application name.</td>
</tr>
<tr>
<td></td>
<td>RETURNS STRING</td>
</tr>
<tr>
<td><code>getChildCount()</code></td>
<td>Retrieve the number of running child applications.</td>
</tr>
<tr>
<td></td>
<td>RETURNS INTEGER</td>
</tr>
<tr>
<td><code>getSessionId()</code></td>
<td>Get the current session identifier.</td>
</tr>
<tr>
<td></td>
<td>RETURNS STRING</td>
</tr>
<tr>
<td><code>getState()</code></td>
<td>Retrieve the application state.</td>
</tr>
<tr>
<td></td>
<td>RETURNS STRING</td>
</tr>
<tr>
<td><code>getStatistics()</code></td>
<td>Retrieve the test session statistics.</td>
</tr>
<tr>
<td></td>
<td>RETURNS Statistics</td>
</tr>
<tr>
<td><code>notifyCheckFailure()</code></td>
<td>Report a check failure.</td>
</tr>
<tr>
<td>(message STRING)</td>
<td></td>
</tr>
<tr>
<td><code>showStatistics()</code></td>
<td>Show test statistics.</td>
</tr>
<tr>
<td></td>
<td>RETURNS Statistics</td>
</tr>
</tbody>
</table>
### Table 33: Methods for the FrontCallRequest

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>(r FrontCallRequest) getFunctionName()</td>
<td>Returns the front call name.</td>
</tr>
<tr>
<td>RETURNS STRING</td>
<td></td>
</tr>
<tr>
<td>(r FrontCallRequest) getModuleName()</td>
<td>Returns the front call module name.</td>
</tr>
<tr>
<td>RETURNS STRING</td>
<td></td>
</tr>
<tr>
<td>(r FrontCallRequest) getParameterCount()</td>
<td>Returns the parameter count to the front call.</td>
</tr>
<tr>
<td>RETURNS INTEGER</td>
<td></td>
</tr>
<tr>
<td>(r FrontCallRequest) getParameterValue(index INTEGER)</td>
<td>Returns the parameter values to the front call.</td>
</tr>
<tr>
<td>RETURNS STRING</td>
<td></td>
</tr>
<tr>
<td>(r FrontCallRequest) getReturnCount()</td>
<td>Returns the return values count to the front call.</td>
</tr>
<tr>
<td>RETURNS INTEGER</td>
<td></td>
</tr>
<tr>
<td>Method</td>
<td>Description</td>
</tr>
<tr>
<td>--------</td>
<td>-------------</td>
</tr>
<tr>
<td><code>(a FrontCallAnswer) functionNotFound()</code></td>
<td>Initializes a front call answer object with &quot;Function not found&quot; error.</td>
</tr>
<tr>
<td><code>(a FrontCallAnswer) moduleNotFound()</code></td>
<td>Initializes a front call answer object with &quot;Module not found&quot; error.</td>
</tr>
<tr>
<td><code>(a FrontCallAnswer) notProcessed()</code></td>
<td>Leave the front call unprocessed.</td>
</tr>
<tr>
<td><code>(a FrontCallAnswer) returnInteger(v INT)</code></td>
<td>Adds an integer return value.</td>
</tr>
<tr>
<td><code>(a FrontCallAnswer) returnString(v STRING)</code></td>
<td>Adds a string return value.</td>
</tr>
<tr>
<td><code>(a FrontCallAnswer) stackError()</code></td>
<td>Initializes a front call answer object with &quot;Stack error&quot; error.</td>
</tr>
<tr>
<td><code>(a FrontCallAnswer) success()</code></td>
<td>Initializes a front call answer object with SUCCESS status.</td>
</tr>
<tr>
<td><code>(a FrontCallAnswer) userError(errorMessage STRING)</code></td>
<td>Initializes a front call answer object with custom error and error message.</td>
</tr>
</tbody>
</table>
Genero Ghost Client Types

The Genero BDL API for GGC provides types for testing.
Table 35: Genero Ghost Client - BDL Types

<table>
<thead>
<tr>
<th>Types</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>PUBLIC TYPE FrontCallAnswer</td>
<td>The FrontCallAnswer type defines a record for retrieving the result of a front call.</td>
</tr>
<tr>
<td>on page 75 RECORD type STRING, status STRING, errorMessage STRING, returnValues DYNAMIC ARRAY OF RECORD isNull BOOLEAN, value STRING, index INTEGER, dataType STRING END RECORD END RECORD</td>
<td></td>
</tr>
<tr>
<td>PUBLIC TYPE FrontCallRequest</td>
<td>The FrontCallRequest type defines a record for retrieving front call request details.</td>
</tr>
<tr>
<td>on page 75 RECORD frontCall RECORD moduleName STRING, functionName STRING, paramCount INTEGER, returnCount INTEGER, parameters DYNAMIC ARRAY OF RECORD dataType STRING, isNull BOOLEAN, value STRING END RECORD END RECORD, errorMessage STRING END RECORD</td>
<td></td>
</tr>
<tr>
<td>PUBLIC TYPE Message</td>
<td>The Message type defines a record for retrieving errors and messages.</td>
</tr>
<tr>
<td>on page 76 RECORD isErrorMessage INTEGER, message STRING END RECORD</td>
<td></td>
</tr>
<tr>
<td>PUBLIC TYPE Statistics</td>
<td>The statistics type defines a record for retrieving test result statistics.</td>
</tr>
<tr>
<td>on page 76 RECORD sessionId STRING, bytesSent STRING, bytesReceived INTEGER, scenarioCount INTEGER, scenarioFailed INTEGER, downloadCount INTEGER, downloadFailed INTEGER, downloadBytes INTEGER, sessionDuration RECORD startTime BIGINT, endTime BIGINT END RECORD, downloadFailures DYNAMIC ARRAY OF STRING, errors DYNAMIC ARRAY OF RECORD status STRING, detailMessage STRING END RECORD, checkFailures DYNAMIC ARRAY OF RECORD fileName STRING, lineNo INTEGER, message STRING END RECORD</td>
<td></td>
</tr>
</tbody>
</table>
**FrontCallAnswer**

The FrontCallAnswer type defines a record for retrieving the result of a front call.

**Syntax**

```plaintext
PUBLIC TYPE FrontCallAnswer on page 75 RECORD
  type STRING,
  status STRING,
  errorMessage STRING,
  returnValues DYNAMIC ARRAY OF RECORD
    isNull BOOLEAN,
    value STRING,
    index INTEGER,
    dataType STRING
  END RECORD
END RECORD
```

1. **type** is the name of the front call type
2. **status** is an error status string
3. **errorMessage** holds details of an error message
4. The **returnValues** array holds the list of the function's return values:
   a. **isNull** if set, a null value is allowed
   b. **value** holds the return value
   c. **index** holds the index position
   d. **dataType** is the data type of the return value

**Usage**

It provides an API to inspect a front call answer; typically the module, and the function status, error type, and return values. A variable of the type `FrontCallAnswer` must be defined, for example:

```plaintext
DEFINE fcAnswer FrontCallAnswer
...
CALL fcAnswer.moduleNotFound()
```

**FrontCallRequest**

The FrontCallRequest type defines a record for retrieving front call request details.

**Syntax**

```plaintext
PUBLIC TYPE FrontCallRequest on page 75 RECORD
  frontCall RECORD
    moduleName STRING,
    functionName STRING,
    paramCount INTEGER,
    returnCount INTEGER,
    parameters DYNAMIC ARRAY OF RECORD
      dataType STRING,
      isNull BOOLEAN,
      value STRING
    END RECORD
  END RECORD,
  errorMessage STRING
END RECORD
```

1. **moduleName** is the name of the front call module
2. **functionName** is the name of the front call function
3. `paramCount` holds the number of function parameters
4. `returnCount` holds the number of values returned by the function
5. The `parameters` array holds the list of the function parameters and their details in the record:
   a. `dataType` is the data type of the parameter
   b. `isNull` if set, a null value is allowed
   c. `value` holds the parameter value
6. `errorMessage` contains details of an error message

Usage

It provides an API to inspect a front call request; typically the module name, the function name, parameters, etc. A variable of the type `FrontCallRequest` must be defined to call these methods, for example:

```gen
DEFINE fcRequest FrontCallRequest
DEFINE moduleName STRING
...
LET moduleName = fcRequest.getModuleName()
```

Message

The `Message` type defines a record for retrieving errors and messages.

Syntax

```gen
PUBLIC TYPE Message on page 76 RECORD
    isErrorMessage INTEGER,
    message STRING
END RECORD
```

1. `isErrorMessage` is set to true, if the message describes an error.
2. `Message` contains the details of the message.

Usage

You can use it to retrieve messages or errors in an application testing the `MESSAGE` or `ERROR` instructions. A variable of the type `Message` must be defined, for example, calling the action `getMessage()`:

```gen
DEFINE msg ggc.Message
CALL ggc.getMessage() RETURNING msg.*
DISPLAY msg.message
```

Statistics

The `Statistics` type defines a record for retrieving test result statistics.

Syntax

```gen
PUBLIC TYPE Statistics on page 76 RECORD
    sessionId STRING,
    bytesSent STRING,
    bytesReceived INTEGER,
    scenarioCount INTEGER,
    scenarioFailed INTEGER,
    downloadCount INTEGER,
    downloadFailed INTEGER,
    downloadBytes INTEGER,
    sessionDuration RECORD
    startTime BIGINT,
```
1. The session record stores the session statistics:
   a. sessionId is the test session identifier
   b. bytesSent is the amount of bytes sent
   c. bytesReceived is the amount of bytes received
   d. scenarioCount is the number of executed scenarios
   e. scenarioFailed is the number of failed scenarios
   f. downloadCount is the number of files downloaded
   g. downloadFailed is the number of file download failures
   h. downloadBytes is the amount of bytes in file downloads
   i. The sessionDuration record stores details of the session duration
      1. startTime is the date the session started (seconds since epoch)
      2. endTime is the date the session ended (seconds since epoch)
   j. The downloadFailures stores an array of failed download URLs
   k. The errors array stores records of the errors reported:
      a. status is an error status string
      b. detailMessage is the error message
   l. The checkFailures stores an array of records with a list of failed checks:
      1. The fileName is the file name where the check failure occurred
      2. The lineNo is the line number in the file
      3. The message is the details of the error

Usage

You can use it to retrieve statistics after application testing is finished. A variable of the type Statistics must be defined, for example, calling the action getStatistics():

```sql
DEFINE stats ggc.Statistics
CALL ggc.getStatistics() RETURNING stats.*
CALL ggc.showStatistics(stats.*)
```
Actions and keys
The Genero BDL API for GGC provides functions for testing with actions and keys.

Table 36: Functions testing actions and keys

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>action(name STRING)</td>
<td>Execute an action by name.</td>
</tr>
<tr>
<td>key(name STRING)</td>
<td>Send a key by name.</td>
</tr>
</tbody>
</table>

**action()**
Execute an action by name.

**Syntax**

```plaintext
action(
    name STRING
)
```

1. `name` is the name of the action to execute.

**Usage**
You can use it to call an action by name in an application to test its action in a scenario. For example, calling the action `cancel`.

```plaintext
CALL ggc.action("cancel")
```

**key()**
Send a key by name.

**Syntax**

```plaintext
key(
    name STRING
)
```

1. `name` is the name of the key to send.

**Usage**
You can use it to send a key in an application to test its action in a scenario. For example, calling the tab key.

```plaintext
CALL ggc.key("tab")
```
Form field interaction

The Genero BDL API for GGC provides functions for interacting with simple form fields.

### Table 37: Functions for interacting with simple form fields

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>setFieldValue()</code></td>
<td>Set value in a form field.</td>
</tr>
</tbody>
</table>

#### `setFieldValue()`
Set value in a form field.

**Syntax**

```plaintext
setFieldValue(
    fieldName STRING,
    value STRING )
```

1. `fieldName` is the name of the form field to set the focus on.
2. `value` is the new value to set in a form field.

**Usage**
You can use it to set the value on a field in an application form to test this action in a scenario. For example:

```plaintext
CALL ggc.setFieldValue("msg", "Hello World!")
```

#### `setFocus()`
Set focus on a field.

**Syntax**

```plaintext
setFocus(
    fieldName STRING )
```

1. `fieldName` is the name of the form field to set the focus on.

**Usage**
You can use it to set the focus on a field in an application form to test its action in a scenario. For example, setting the focus on the "firstname" field.

```plaintext
CALL ggc.setFocus("firstname")
```
**setValue()**
Set value in the current field.

**Syntax**

```
setValue(
  value  STRING )
```

1. `value` is the new value to set in the field.

**Usage**

You can use it to set the value on a field in an application form to test its action in a scenario. For example:

```
CALL ggc.setValue("Hello World!")
```
Table interaction

The Genero BDL API for GGC provides functions for interacting with tables.
### Table 38: Functions for interacting with tables

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>getColumnValue(tableName STRING, columnName STRING, row INTEGER)</code></td>
<td>Get the column value of a table, tree, or screen record at the specified row.</td>
</tr>
<tr>
<td><code>getColumnValues(tableName STRING, columnName STRING)</code></td>
<td>Get column values of a table, tree, or screen record.</td>
</tr>
<tr>
<td><code>hideTableColumn(tableName STRING, columnName STRING)</code></td>
<td>Hide a table column.</td>
</tr>
<tr>
<td><code>setCellFocus(tableName STRING, columnName STRING, row INTEGER)</code></td>
<td>Select a cell in a table.</td>
</tr>
<tr>
<td><code>setRowFocus(tableName STRING, row INTEGER)</code></td>
<td>Select a table row.</td>
</tr>
<tr>
<td><code>setRowSelection(tableName STRING, selectionMode STRING, startIndex INTEGER, endIndex INTEGER)</code></td>
<td>Update a multiple row selection state.</td>
</tr>
<tr>
<td><code>setTableOffset(tableName STRING, pageSize INTEGER)</code></td>
<td>Set the table page size offset value.</td>
</tr>
<tr>
<td><code>setTableSize(tableName STRING, size INTEGER)</code></td>
<td>Set the table size.</td>
</tr>
<tr>
<td><code>showTableColumn(tableName STRING, columnName STRING)</code></td>
<td>Show a hidden table column.</td>
</tr>
<tr>
<td><code>sortTable(tableName STRING, columnName STRING, sortType STRING)</code></td>
<td>Sort a table.</td>
</tr>
</tbody>
</table>
getColumnValue()
Get the column value of a table, tree, or screen record at the specified row.

Syntax
```
getColumnValue(  
    tableName  STRING,  
    columnName STRING,  
    row    INTEGER )  
RETURNS STRING
```

1. `tableName` is the name of a table.
2. `columnName` is the name of the column.
3. `row` is the row index of a table.

Usage
You can use it to return the value of a column at a row of a table, tree, or a visible row of a screen record to test this action in a scenario. For example:

```
DEFINE item STRING
LET item = ggc.getColumnValue("sr_prices", "name", 1)
```

Related concepts
getColumnValues() on page 83
Get column values of a table, tree, or screen record.

getColumnValues()
Get column values of a table, tree, or screen record.

Syntax
```
getColumnValues(  
    tableName  STRING,  
    columnName STRING )  
RETURNS DYNAMIC ARRAY OF STRING
```

1. `tableName` is the name of a table.
2. `columnName` is the name of the column.

Usage
You can use it to return the values of a column of a table, tree, or visible rows of a screen record to test this action in a scenario. For example:

```
DEFINE items DYNAMIC ARRAY OF STRING
LET items = ggc.getColumnValues("sr_prices", "name")
```

Related concepts
getColumnValue() on page 83
Get the column value of a table, tree, or screen record at the specified row.

**hideTableColumn()**
Hide a table column.

**Syntax**

```plaintext
hideTableColumn(
   tableName STRING,
   columnName STRING)
```

1. `tableName` is the name of the table.
2. `columnName` is the name of the table column.

**Usage**
You can use it to hide a table column to test this action in a scenario. For example:

```plaintext
CALL ggc.hideTableColumn("items", "description")
```

**setCellFocus()**
Select a cell in a table.

**Syntax**

```plaintext
setCellFocus(
   tableName STRING,
   columnName STRING,
   row INTEGER)
```

1. `tableName` is the name of the table.
2. `columnName` is the name of a column.
3. `row` is the index number of a row.

**Usage**
You can use it to select a cell in a table to test this action in a scenario. For example:

```plaintext
CALL ggc.setCellFocus("items", "description", 100)
```

**setRowFocus()**
Select a table row.

**Syntax**

```plaintext
setRowFocus(
   tableName STRING,
   row INTEGER)
```

1. `tableName` is the name of the table.
2. `row` is the index number of a row.
Usage
You can use it to select a row in a table to test this action in a scenario. For example:

```plaintext
CALL ggc.setRowFocus("items", 100)
```

`setRowSelection()`
Update a multiple row selection state.

Syntax
```
setRowSelection(
    tableName STRING,
    selectionMode STRING,
    startIndex INTEGER,
    endIndex INTEGER)
```

1. `tableName` is the name of the table.
2. `selectionMode` is the method of selection specified by these values.
   
   `selectionMode` values:
   
   - `ggc.MRS_SET` - Clear and set a new row selection.
   - `ggc.MRS_UNSET` - Unset a row selection.
   - `ggc.MRS_EXSET` - Extend a row selection.

3. `startIndex` is the index row number to start the selection.
4. `endIndex` is the index row number to end the selection.

Usage
You can use it to perform a selection on multiple rows in a table. For example:

**Clear and set a row selection.**

```plaintext
CALL ggc.setRowSelection("items", ggc.MRS_SET, 10, 20)
```

**Extend a row selection**

```plaintext
CALL ggc.setRowSelection("items", ggc.MRS_EXSET, 25, 26)
```

`setTableOffset()`
Set the table page size offset value.

Syntax
```
setTableOffset(
    tableName STRING,
    pageSize INTEGER )
```

1. `tableName` is the name of the table.
2. `pageSize` is the number of rows, as an offset value, from the table's current start row index.

Usage
You can use it to set the page size of a table in an application to test this action in a scenario. For example:

```plaintext
CALL ggc.setTableOffset("items", 100)
```
Related concepts

getTableOffset() on page 98
Get the table's current offset value.

setTableSize()
Set the table size.

Syntax

```plaintext
setTableSize(
    tableName  STRING,
    size       INTEGER)
```

1. `tableName` is the name of the table.
2. `size` is the new size to set the table.

Usage

You can use it to set the size of a table in an application to test this action in a scenario. For example:

```plaintext
CALL ggc.setTableSize("items", 100)
```

showTableColumn()
Show a hidden table column.

Syntax

```plaintext
showTableColumn(
    tableName  STRING,
    columnName STRING)
```

1. `tableName` is the name of the table.
2. `columnName` is the name column.

Usage

You can use it to show a hidden table column to test this action in a scenario. For example:

```plaintext
CALL ggc.showTableColumn("items", "description")
```

sortTable()
Sort a table.

Syntax

```plaintext
sortTable(
    tableName  STRING,
    columnName STRING,
    sortType   STRING)
```

1. `tableName` is the name of the table.
2. `columnName` is the name of a column.
3. `sortType` is the sort order

   `sortType` values:
   - `ggc.SORT_ASCENDING` - Sort ascending order.
   - `ggc.SORT_DESCENDING` - Sort descending order.
Usage
You can use it to sort a table to test this action in a scenario. For example:

```sql
CALL ggc.sortTable("items", "description", ggc.SORT_DESCENDING)
```

Tree view interaction
The Genero BDL API for GGC provides functions for interacting with tree views.

Important:
All Table interaction on page 81 functions can be used with tree views. This section deals with tree view only functions.

Table 39: Functions for interacting with tree views

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>collapseTree</code></td>
<td>Collapse a node (row) in a tree view.</td>
</tr>
<tr>
<td><code>expandTree</code></td>
<td>Expand a node (row) in a tree view.</td>
</tr>
</tbody>
</table>

**collapseTree()**
Collapse a node (row) in a tree view.

**Syntax**

```sql
collapseTree(
  treeName STRING,
  row INTEGER )
```

1. `treeName` is the name of the tree view.
2. `row` is the index row number.

**Usage**
You can use it to collapse a selected node in a specified tree view to test this action in a scenario. For example:

```sql
CALL ggc.collapseTree("items", 100)
```

**expandTree()**
Expand a node (row) in a tree view.

**Syntax**

```sql
expandTree(
  treeName STRING,
  row INTEGER )
```
1. *treeName* is the name of the tree view.
2. *row* is the index row number.

**Usage**

You can use it to expand a selected node in a specified tree view to test this action in a scenario. For example:

```
CALL ggc.expandTree("items", 100)
```
Retrieve information and data

The Genero BDL API for GGC provides functions for retrieving information and data when testing applications.

Table 40: Functions for retrieving information and data

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>getFieldValue(fieldName STRING) RETURNS STRING</code></td>
<td>Return the value in a specified form field.</td>
</tr>
<tr>
<td><code>getFieldValues(name STRING) RETURNS DYNAMIC ARRAY OF STRING</code></td>
<td>Return a list of values from a table, tree, or screen record.</td>
</tr>
<tr>
<td><code>getFocus()</code> <code>RETURNS STRING</code></td>
<td>Return the name of the current focused element.</td>
</tr>
<tr>
<td><code>getFormName()</code> <code>RETURNS STRING</code></td>
<td>Get the current form name.</td>
</tr>
<tr>
<td><code>getFormTitle()</code> <code>RETURNS STRING</code></td>
<td>Get the current form title.</td>
</tr>
<tr>
<td><code>getValue()</code> <code>RETURNS STRING</code></td>
<td>Return the value in the current field.</td>
</tr>
<tr>
<td><code>getValues()</code> <code>RETURNS DYNAMIC ARRAY OF STRING</code></td>
<td>Returns the values of the current row of the current table, tree, or matrix.</td>
</tr>
<tr>
<td><code>getWidgetType(fieldName STRING) RETURNS STRING</code></td>
<td>Return the widget type of the specified field.</td>
</tr>
<tr>
<td><code>getWindowName()</code> <code>RETURNS STRING</code></td>
<td>Get the current window name.</td>
</tr>
<tr>
<td><code>getWindowTitle()</code> <code>RETURNS STRING</code></td>
<td>Get the current window title.</td>
</tr>
</tbody>
</table>

getFieldValue()

Return the value in a specified form field.

Syntax

```
getFieldValue(
```
fieldName STRING)
RETURNS STRING

1. fieldName is the name of the field.

Usage
You can use it to return the value of a specified field in an application form to test this action in a scenario. It cannot be used in DISPLAY ARRAY. For example:

DISPLAY "The last name is " || ggc.getValue("last_name")

Related concepts
setFieldValue() on page 79
Set value in a form field.

getFieldValues()
Return a list of values from a table, tree, or screen record.

Syntax

```
ggetFieldValues (  
  name STRING)  
  RETURNS DYNAMIC ARRAY OF STRING
```

1. name is the name of the table, tree, or screen record.

Usage
You can use it to return a list of values in the current row of a table, tree, or screen record. For example:

CALL ggc.getFieldValues("sr_table") RETURNING values

getFocus()
Return the name of the current focused element.

Syntax

```
getFocus()  
  RETURNS STRING
```

Usage
You can use it to return the name of the focused element to test this action in a scenario. It can be a simple form field, a table, or a tree. For example:

DISPLAY "The focus is now in " || ggc.getFocus()

Related concepts
setFocus() on page 79
Set focus on a field.

**getFormName()**
Get the current form name.

**Syntax**

```
getFormName()
RETURNS STRING
```

**Usage**

You can use it to return the name of the current application form to test this action in a scenario. For example:

```
LET formName = ggc.getFormName()
```

**getFormTitle()**
Get the current form title.

**Syntax**

```
getFormTitle()
RETURNS STRING
```

**Usage**

You can use it to return the title of the current application form to test this action in a scenario. For example:

```
LET formTitle = ggc.getFormTitle()
```

**getValue()**
Return the value in the current field.

**Syntax**

```
getValue()
RETURNS STRING
```

**Usage**

You can use it to return the value of the focused element in an application form to test this action in a scenario. It can be a simple form field, or a table cell. It can not be used in DISPLAY ARRAY. For example:

```
DISPLAY "The value of the current form field is " || ggc.getValue()
```

**Related concepts**

setValue() on page 80
Set value in the current field.

**getValues()**
Returns the values of the current row of the current table, tree, or matrix.

**Syntax**

```plaintext
getValues ()
    RETURNS DYNAMIC ARRAY OF STRING
```

**Usage**
You can use it to return a list of values in the current row of the current table, tree, or matrix. Use it to test this action in a scenario. For example:

```plaintext
DEFINE i INTEGER
    DEFINE values DYNAMIC ARRAY OF STRING
    CALL ggc.getValues() RETURNING values
    DISPLAY "Row values:"
    FOR i = 1 TO values.getLength()
        DISPLAY "va", i, " = ", values[i]
    END FOR
```

**getWidgetType()**
Return the widget type of the specified field.

**Syntax**

```plaintext
getWidgetType (fieldName STRING)
    RETURNS STRING
```

1. *fieldName* is the name of the field.

**Usage**
You can use it to return the widget type of a specified field to test this action in a scenario. The widget type returned can be:

- formfield
- table
- tree
- matrix
- scrollarea

For example:

```plaintext
DISPLAY "The widget type is " || ggc.getWidgetType("tbl1")
```

**getWindowName()**
Get the current window name.

**Syntax**

```plaintext
getWindowName ()
    RETURNS STRING
```
Usage
You can use it to return the name of an application window to test this action in a scenario. For example:

```
LET windowName = ggc.getWindowName()
```

**getWindowTitle()**
Get the current window title.

**Syntax**
```
getWindowTitle() RETURNS STRING
```

Usage
You can use it to return the title of an application window to test this action in a scenario. For example:

```
LET windowTitle = ggc.getWindowTitle()
```

Retrieve MESSAGE and ERROR
The Genero BDL API for GGC provides functions for retrieving messages and errors when testing applications.

The following functions retrieve messages or error messages from the current instructions (MESSAGE or ERROR) to the user allowing you to test these interactions.

The value is returned in a `ggc.Message` record type.

**Table 41: Functions for retrieving messages and errors**

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>getError()</code></td>
<td>Return the current error message.</td>
</tr>
<tr>
<td><code>getMessage()</code></td>
<td>Return the current message.</td>
</tr>
</tbody>
</table>

**getError()**
Return the current error message.

**Syntax**
```
getError() RETURNS ggc.Message
```

In the return the `ggc.isErrorMessage` field of the `Message` on page 76 message record should be TRUE.

**Usage**
You can use it to retrieve the error message from the current (ERROR) instruction to the user allowing you to test this interaction in a scenario.

For more information on BDL ERROR statements, see the `ERROR` topic in *Genero Business Development Language User Guide* for details.
You must declare a variable of type `ggc.message` for the return. For example:

```
DEFINE msg ggc.Message
CALL ggc.getError() RETURNING msg.*
DISPLAY msg.message
```

**CAUTION:** Avoid writing code like this:

```
LET msg = ggc.getError()
DISPLAY ggc.getError().message
```

**Related concepts**

**getMessage()** on page 94
Return the current message.

**getMessage()**
Return the current message.

**Syntax**

```
getMessage()
RETURNS ggc.Message
```

In the return the `ggc.isErrorMessage` field of the `Message` on page 76 record should be `FALSE`.

**Usage**

You can use it to return the message from the current (MESSAGE) instruction to the user allowing you to test this interaction in a scenario.

For more information on MESSAGE statements, see the MESSAGE topic in *Genero Business Development Language User Guide* for details.

You must declare a variable of type `ggc.message` for the return. For example:

```
DEFINE msg ggc.Message
CALL ggc.getMessage() RETURNING msg.*
DISPLAY msg.message
```

**CAUTION:** Avoid writing code like this:

```
LET msg = ggc.getMessage()
DISPLAY ggc.getMessage().message
```

**Related concepts**

**getError()** on page 93
Retrieve action state

The Genero BDL API for GGC provides functions for retrieving action state when testing applications.

Table 42: Functions for retrieving action state

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>isActionActive()</td>
<td>Return the state of the specified action.</td>
</tr>
</tbody>
</table>

**isActionActive()**
Return the state of the specified action.

**Syntax**

```plaintext
isActionActive(
    name STRING
) RETURNS BOOLEAN
```

The return is TRUE if the action is active, FALSE if the action is not active.

1. **Name** is the action name.

**Usage**

You can use it to return the state of the specified action to test this interaction in a scenario. If the action is not found, ggc.statusCode is set to ggc.ACTION_NOT_FOUND.

For example:

```plaintext
IF ggc.isActionActive("cancel") THEN
    DISPLAY "The action cancel is active"
ELSE
    CALL ggc.notifyCheckFailure("The action cancel is not active.!!")
END IF
```
**Table information**

The Genero BDL API for GGC provides functions for retrieving information about tables.

**Table 43: Functions for retrieving information about tables**

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>getColumnCount(tableName STRING)</code></td>
<td>Get the number of columns in a table.</td>
</tr>
<tr>
<td><code>getColumnName(tableName STRING, idx INTEGER)</code></td>
<td>Get a table column name.</td>
</tr>
<tr>
<td><code>getCurrentColumn(tableName STRING)</code></td>
<td>Get the current column in a table.</td>
</tr>
<tr>
<td><code>getCurrentRow(tableName STRING)</code></td>
<td>Get the current row of the table.</td>
</tr>
<tr>
<td><code>getTableOffset(tableName STRING)</code></td>
<td>Get the table's current offset value.</td>
</tr>
<tr>
<td><code>getTableSize(tableName STRING)</code></td>
<td>Get the table size.</td>
</tr>
</tbody>
</table>

**getRowCount()**

Get the number of columns in a table.

**Syntax**

```java
getColumnCount (tableName STRING )
RETURNS INTEGER
```

1. `tableName` is the name of the table.

**Usage**

You can use it to get a count value of the number of columns in a table to test this action in a scenario. For example:

```java
DISPLAY SFMT("Items has %1 columns", ggc.getColumnCount("items"))
```
**getColumnName()**
Get a table column name.

**Syntax**

```plaintext
getColumnName(
    tableName  STRING,
    idx  INTEGER  )
RETURNS STRING
```

1. `tableName` is the name of the table.
2. `idx` is the column index number

**Usage**
You can use it to retrieve the column name from a table to test this action in a scenario. For example:

```plaintext
FOR i = 1 TO ggc.getColumnCount("items")
    DISPLAY SFMT("Column %1: %2", i, ggc.getColumnName(items, i))
END FOR
```

**getCurrentColumn()**
Get the current column in a table.

**Syntax**

```plaintext
getCurrentColumn(
    tableName  STRING  )
RETURNS INTEGER
```

1. `tableName` is the name of the table.

**Usage**
You can use it to retrieve the current column of a table to test this action in a scenario. For example:

```plaintext
DISPLAY "The focus is in the column ", ggc.getCurrentColumn("items")
```

**getCurrentRow()**
Get the current row of the table.

**Syntax**

```plaintext
getCurrentRow(
    tableName  STRING  )
RETURNS INTEGER
```

1. `tableName` is the name of the table.

**Usage**
You can use it to retrieve the current row, as the row index, in a table to test this action in a scenario. For example:

```plaintext
DISPLAY "The current row of the items table is ", ggc.getCurrentRow("items")
```
**getTableOffset()**
Get the table's current offset value.

**Syntax**

```bdl
getTableOffset(
    tableName  STRING)
RETURNS INTEGER
```

Returns the table's current start row index.

1. `tableName` is the name of the table.

**Usage**
You can use it to retrieve the table offset, as the start row index, to test this action in a scenario. For example:

```bdl
DISPLAY "First visible row is ", ggc.getTableOffset("items")
```

**Related concepts**
- `setTableOffset()` on page 85
  Set the table page size offset value.
- `getTableSize()`
  Get the table size.

**Syntax**

```bdl
getTableSize(
    tableName  STRING)
RETURNS INTEGER
```

Returns the table size.

1. `tableName` is the name of the table.

**Usage**
You can use it to retrieve the size of a table in an application to test this action in a scenario. For example:

```bdl
FOR i = 1 TO ggc.getTableSize("items")
    CALL ggc.setCellFocus("items", "description", i)
    DISPLAY SFMT("Item %1: %s", i, ggc getValue())
END FOR
```

**Related concepts**
- `setTableSize()` on page 86
  Set the table size.

**Session information**
The Genero BDL API for GGC provides functions for retrieving information about sessions.
The following functions allow you to retrieve information from the current session.

Functions `getStatistics()` on page 100 or `showStatistics()` on page 101 return values in a `ggc.Statistics` record type.
### Table 44: Functions for retrieving session information

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>getApplicationName()</code></td>
<td>Retrieve the application name.</td>
</tr>
<tr>
<td><code>getChildCount()</code></td>
<td>Retrieve the number of running child applications.</td>
</tr>
<tr>
<td><code>getSessionId()</code></td>
<td>Get the current session identifier.</td>
</tr>
<tr>
<td><code>getState()</code></td>
<td>Retrieve the application state.</td>
</tr>
<tr>
<td><code>getStatistics()</code></td>
<td>Retrieve the test session statistics.</td>
</tr>
<tr>
<td><code>notifyCheckFailure()</code></td>
<td>Report a check failure.</td>
</tr>
<tr>
<td><code>showStatistics()</code></td>
<td>Show test statistics.</td>
</tr>
</tbody>
</table>

**getApplicationName()**

Retrieve the application name.

**Syntax**

```java
getApplicationName ()
RETURNS STRING
```

**Usage**

You can use it to return the application name to test this action in a scenario. For example:

```java
DISPLAY "The application name is " || ggc.getApplicationName()
```

**getChildCount()**

Retrieve the number of running child applications.

**Syntax**

```java
getChildCount ()
RETURNS INTEGER
```

Returns the application child count.
**Usage**

You can use it get the number of child applications running during a test session. For example:

```plaintext
DISPLAY "The application child count is " || getChildCount()
```

**getSessionId()**

Get the current session identifier.

**Syntax**

```plaintext
getSessionId()  
RETURNS STRING
```

**Usage**

You can use it to return the id of a session. This identifier can be used to locate log files in GAS / JGAS and GGC. Log file names will include this session identifier. For example:

```plaintext
DISPLAY "The current session id is " || ggc.getSessionId()
```

**getState()**

Retrieve the application state.

**Syntax**

```plaintext
getState()  
RETURNS STRING
```

Returns the application state.

**Usage**

You can use it get the application state during a test session. The application state can be:

- **INIT** - Client is initialized
- **START** - Client is starting
- **PROCESSING** - VM processing
- **INTERACTIVE** - VM is interactive
- **FUNCTION_CALL** - Processing a function call
- **ENDED** - Client has ended

For example:

```plaintext
DISPLAY "The application state is " || getState()
```

**getStatistics()**

Retrieve the test session statistics.

**Syntax**

```plaintext
getStatistics()  
RETURNS Statistics
```

Returns the test session statistics
Usage
You can use it when the session ends to collect test statistics generated. You must declare a variable of type `ggc.Statistics` for the return. For example:

```sql
DEFINE stats ggc.Statistics
CALL ggc.getStatistics() RETURNING stats.*
CALL ggc.showStatistics(stats.*)
```

Related concepts
showStatistics() on page 101
Show test statistics.

notifyCheckFailure()
Report a check failure.

Syntax

```sql
notifyCheckFailure ( message STRING )
```

1. `message` is the failure message to register.

Usage
You can use it to notify the scenario user of a check failure. If you write a check and expect a failure to be recorded and reported at the end, call this function when your check fails. The failure will be reported with file and line number information. At the end of the scenario all failures will be listed, formatted as:

```
filename:lineno:error:message
```

For example:

```
/home/f4gl/app/myapp.4gl:82:error:(GGC-2) The action 'hello' is not found.
```

Failed checks generate error numbers. GGC error numbers (for example GGC-2), are displayed in a way that makes them clickable in Genero Studio so the file and line where the error occurred can be found. For this reason, you need to provide error numbers for the check failures you report. Calling this function does not stop the scenario. For example:

```sql
DEFINE msg ggc.Message
CALL ggc.getMessage() RETURNING msg.*
DISPLAY msg.message
IF msg.message != "User successfully created." THEN
   CALL notifyCheckFailure("(GGC-2) Expected 'User successfully created.',
got: '' || msg.message || ''")
END IF
```

showStatistics()
Show test statistics.

Syntax

```sql
showStatistics ()
RETURNS Statistics
```
Returns the session statistics.

Usage
You can use it to show test statistics. You must declare a variable of type `ggc.Statistics`. For example:

```plaintext
DEFINE stats ggc.Statistics
CALL ggc.getStatistics() RETURNING stats.*
CALL ggc.showStatistics(stats.*)
```

Related concepts
`getStatistics()` on page 100
Retrieve the test session statistics.

Front call request
The Genero BDL API for GGC provides methods for front call requests.

The `ggc.FrontCallRequest` type provides an API to inspect a front call request record; typically the module name, the function name, parameters, etc.

This section documents these front call related APIs. For an example of Genero BDL code using these methods, see `Example: custom front call (BDL)` on page 47. This example features use of the methods: `getModuleName()` on page 103, `getFunctionName()` on page 103, and `getParameterValue()` on page 104 as you would use these in a front call implementation.

Table 45: Methods for the FrontCallRequest

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>(r FrontCallRequest) getFunctionName()</code></td>
<td>Returns the front call name.</td>
</tr>
<tr>
<td>RETURNS STRING</td>
<td></td>
</tr>
<tr>
<td><code>(r FrontCallRequest) getModuleName()</code></td>
<td>Returns the front call module name.</td>
</tr>
<tr>
<td>RETURNS STRING</td>
<td></td>
</tr>
<tr>
<td><code>(r FrontCallRequest) getParameterCount()</code></td>
<td>Returns the parameter count to the front call.</td>
</tr>
<tr>
<td>RETURNS INTEGER</td>
<td></td>
</tr>
<tr>
<td><code>(r FrontCallRequest) getParameterValue(index INTEGER)</code></td>
<td>Returns the parameter values to the front call.</td>
</tr>
<tr>
<td>RETURNS STRING</td>
<td></td>
</tr>
<tr>
<td><code>(r FrontCallRequest) getReturnCount()</code></td>
<td>Returns the return values count to the front call.</td>
</tr>
<tr>
<td>RETURNS INTEGER</td>
<td></td>
</tr>
</tbody>
</table>

Related concepts
`Front calls` on page 43
As a developer, you must know how to handle front calls in your test.

**getFunctionName()**
Returns the front call name.

**Syntax**
```c
(r FrontCallRequest) getFunctionName()
RETURNS STRING
```

1. `r` is the front call request object.

**Usage**
A call to this method returns the front call name. If the function is a `standard.feInfo()` front call, the function name returned is "feinfo".

You must declare a variable of type `ggc.FrontCallRequest`. For example:
```c
DEFINE fcRequest FrontCallRequest
DEFINE functionName STRING
...
LET functionName = fcRequest.getFunctionName()
```

**Related reference**
- Default front calls on page 43
Front call implementations that are handled by default by the GGC.

**getModuleName()**
Returns the front call module name.

**Syntax**
```c
(r FrontCallRequest) getModuleName()
RETURNS STRING
```

1. `r` is the front call request object.

**Usage**
A call to this method returns the front call module name. If the function is a `standard.feInfo()` front call, the module name returned is "standard".

You must declare a variable of type `ggc.FrontCallRequest`. For example:
```c
DEFINE fcRequest FrontCallRequest
DEFINE moduleName STRING
...
LET moduleName = fcRequest.getModuleName()
```

**Related reference**
- Default front calls on page 43
Front call implementations that are handled by default by the GGC.

**getParametersCount()**

Returns the parameter count to the front call.

**Syntax**

```
(r FrontCallRequest) getParametersCount()
RETURNS INTEGER
```

1. `r` is the front call request object.

**Usage**

A call to this method returns the function call parameter count. You must declare a variable of type `ggc.FrontCallRequest`. For example:

```
DEFINE fcRequest FrontCallRequest
DEFINE parameterCount INTEGER
...
LET parameterCount = fcRequest.getParameterCount()
```

**Related concepts**

Front calls on page 43

As a developer, you must know how to handle front calls in your test.

**getParameterValue()**

Returns the parameter values to the front call.

**Syntax**

```
(r FrontCallRequest) getParameterValue(index INTEGER)
RETURNS STRING
```

1. `r` is the front call request object.
2. `index` is the parameter index.

**Usage**

A call to this method returns the parameter value at the given index. You must declare a variable of type `ggc.FrontCallRequest`. For example:

```
DEFINE fcRequest FrontCallRequest
DEFINE value STRING
...
LET value = fcRequest.getParameterValue(1)
```

**Related concepts**

Front calls on page 43
As a developer, you must know how to handle front calls in your test.

**getReturnCount()**
Returns the return values count to the front call.

### Syntax

```
(r FrontCallRequest) getReturnCount ()
RETURNS INTEGER
```

1. `r` is the front call request object.

### Usage

A call to this method returns the number of return values the front call expects. You must declare a variable of type `ggc.FrontCallRequest`. For example:

```
DEFINE fcRequest FrontCallRequest
DEFINE returnCount INTEGER
...
LET returnCount = fcRequest.getReturnCount()
```

### Related concepts

- [Front calls](#) on page 43

As a developer, you must know how to handle front calls in your test.

### Front call answer

The Genero BDL API for GGC provides methods for answers to front call requests.

The `ggc.FrontCallAnswer` type provides an API to build responses to function call requests.

This section documents these front call related APIs. For an example of Genero BDL code using these methods, see [Example: custom front call (BDL)](#) on page 47. This example features use of the methods: `notProcessed()` on page 107, `stackError()` on page 109, `userError()` on page 110, and `success()` on page 109 as they are used in a front call implementation.
Table 46: Methods for the FrontCallAnswer

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>(a FrontCallAnswer) functionNotFound()</td>
<td>Initializes a front call answer object with &quot;Function not found&quot; error.</td>
</tr>
<tr>
<td>(a FrontCallAnswer) moduleNotFound()</td>
<td>Initializes a front call answer object with &quot;Module not found&quot; error.</td>
</tr>
<tr>
<td>(a FrontCallAnswer) notProcessed()</td>
<td>Leave the front call unprocessed.</td>
</tr>
<tr>
<td>(a FrontCallAnswer) returnInteger(value INTEGER)</td>
<td>Adds an integer return value.</td>
</tr>
<tr>
<td>(a FrontCallAnswer) returnString(value STRING)</td>
<td>Adds a string return value.</td>
</tr>
<tr>
<td>(a FrontCallAnswer) stackError()</td>
<td>Initializes a front call answer object with &quot;Stack error&quot; error.</td>
</tr>
<tr>
<td>(a FrontCallAnswer) success()</td>
<td>Initializes a front call answer object with SUCCESS status.</td>
</tr>
<tr>
<td>(a FrontCallAnswer) userError(errorMessage STRING)</td>
<td>Initializes a front call answer object with custom error and error message.</td>
</tr>
</tbody>
</table>

Related concepts

Front calls on page 43
As a developer, you must know how to handle front calls in your test.

functionNotFound()
Initializes a front call answer object with "Function not found" error.

Syntax

(a FrontCallAnswer) functionNotFound()

1. a is the front call answer object to initialize.

Usage

You would use this method after a front call request to getFunctionName() on page 103. If an error is returned, the front call chain will stop and the error is returned to the DVM.
You must declare a variable of type `ggc.FrontCallAnswer`. For example:

```plaintext
DEFINE fcAnswer FrontCallAnswer
...
CALL fcAnswer.functionNotFound()
```

### Related concepts
**Front calls** on page 43
As a developer, you must know how to handle front calls in your test.

### moduleNotFound()
Initializes a front call answer object with "Module not found" error.

#### Syntax

```plaintext
(a FrontCallAnswer) moduleNotFound()
```

1. `a` is the front call answer object to initialize.

#### Usage
This method initializes a front call answer with "Module not found" error. If an error is returned, the front call chain will stop and the error is returned to the DVM. You would use this method after a front call request to `getModuleName()` on page 103.

You must declare a variable of type `ggc.FrontCallAnswer`. For example:

```plaintext
DEFINE fcAnswer FrontCallAnswer
...
CALL fcAnswer.moduleNotFound()
```

### Related concepts
**Front calls** on page 43
As a developer, you must know how to handle front calls in your test.

### notProcessed()
Leave the front call unprocessed.

#### Syntax

```plaintext
(a FrontCallAnswer) notProcessed()
```

1. `a` is the front call answer object to initialize.

#### Usage
This method is used when you register multiple front call handlers. The GGC manages one `request` for all your registered handlers. The handlers are linked in a chain. If `notProcessed()` is returned for a handler, the chain continues to the next handler until the requested handler is processed. `notProcessed()` maintains the request chain unbroken.

If, for example, `standard.feInfo("feName")` and `openfile` are two front calls handled, they will be called in the order registered, one after the other. If the request is `openfile`, `notProcessed()` is returned for `feInfo("feName")` (meaning it is skipped, or ignored this time). The request is then provided to the next front call handler - `openfile` in this case.

If success with values or error is returned, the chain stops, the handler has been processed. The last front call handler in the chain is the default front call handler provided by the GGC.
Tip: As handlers are small functions that process one specific front call and are handled in a chain, this makes them flexible enough for reuse with different scenarios.

You must declare a variable of type `ggc.FrontCallAnswer`. Call the method on the front call answer object. For example:

```plaintext
define fcAnswer FrontCallAnswer
... 
call fcAnswer.notProcessed()
```

Related concepts

- Register front call handler on page 44
- Register custom front calls with the front call handler.

Example: custom front call (BDL) on page 47
- Overloading the default `standard.feInfo("feName")` front call.

**returnInteger()**

Adds an integer return value.

**Syntax**

```plaintext
(a FrontCallAnswer) returnInteger(
    value INTEGER)
```

1. `a` is the function call answer object to initialize.
2. `value` is the integer value to return.

**Usage**

This method adds an integer return value. You would use this method after a front call request. After checking for `success()` on page 109, values can be returned.

You must declare a variable of type `ggc.FrontCallAnswer`. For example:

```plaintext
define fcAnswer FrontCallAnswer
... 
# Return two values, '1' and '3'
call fcAnswer.success()
call fcAnswer.returnInteger(1)
call fcAnswer.returnInteger(3)
```

Related concepts

- Front calls on page 43
  - As a developer, you must know how to handle front calls in your test.

**returnString()**

Adds a string return value.

**Syntax**

```plaintext
(a FrontCallAnswer) returnString(
    value STRING)
```

1. `a` is the front call answer object to initialize.
2. `value` is the string value to return.
### Usage

This method adds a string return value. You would use this method after a front call request. After a check with `success()` on page 109, values can be returned.

You must declare a variable of type `ggc.FrontCallAnswer`. For example:

```python
DEFINE fcAnswer FrontCallAnswer
...
# Return two values, 'foo' and 'bar'
CALL fcAnswer.success()
CALL fcAnswer.returnString("foo")
CALL fcAnswer.returnString("bar")
```

### Related concepts

**Front calls** on page 43
As a developer, you must know how to handle front calls in your test.

**stackError()**
Initializes a front call answer object with "Stack error" error.

#### Syntax

```python
(a FrontCallAnswer) stackError()
```

1. `a` is the front call answer object to initialize.

### Usage

A stack error can be returned, for example, when the number of parameters returned in a (getParameterCount() on page 104) call is invalid. If an error is returned, the front call chain will stop and the error is returned to the DVM.

You must declare a variable of type `ggc.FrontCallAnswer`. For example:

```python
DEFINE fcRequest FrontCallRequest
DEFINE fcAnswer FrontCallAnswer
...
IF fcRequest.getParameterCount() != 1 THEN
  CALL fcAnswer.stackError()
END IF
```

### Related concepts

**Front calls** on page 43
As a developer, you must know how to handle front calls in your test.

**success()**
Initializes a front call answer object with SUCCESS status.

#### Syntax

```python
(a FrontCallAnswer) success()
```

1. `a` is the front call answer object to initialize.

### Usage

You would call this method after a front call request. In case of success, the front call can return values using the `ggc.FrontCallAnswer` methods `returnString()` on page 108 or `returnInteger()` on page 108.
You must declare a variable of type `ggc.FrontCallAnswer`. For example:

```java
DEFINE fcAnswer FrontCallAnswer
...
CALL fcAnswer.success()
```

### Related concepts

**Front calls** on page 43  
As a developer, you must know how to handle front calls in your test.

**userError()**  
Initializes a front call answer object with custom error and error message.

#### Syntax

```java
(a FrontCallAnswer) userError(errorMessage STRING)
```

1. `a` is the front call answer object to initialize.
2. `errorMessage` is the error message.

#### Usage

For example, you would call this method after a front call request to `getParameterValue()` on page 104. If an error is returned, the front call chain will stop and the error is returned to the DVM.

You must declare a variable of type `ggc.FrontCallAnswer`. For example:

```java
DEFINE fcRequest FrontCallRequest
DEFINE fcAnswer FrontCallAnswer
...
IF fcRequest.getParameterValue(1) != "foo" THEN
    CALL fcAnswer.userError("Parameter 1 should be foo")
END IF
```

### Related concepts

**Front calls** on page 43  
As a developer, you must know how to handle front calls in your test.

---

### Java API functions for GGC

The Genero Ghost Client provides a variety of Genero Java API functions for use with generating tests in Java.

For details of the Java packages and classes see the javadoc distributed with the Genero Ghost Client in your FGLGWS installation. The javadoc is found in your `GGCDIR/doc/javadoc/` directory. See the help file by launching the `/doc/javadoc/index.html` file in your browser.

---

### HTTP handlers

HTTP handlers can be used to process responses that are not part of the AUI tree. This arises when Single sign on is used. SSO is implemented with a default handler.

The Java API allows for the implementation of HTTP handlers. The interface `com.fourjs.ggc.HttpHandler` supports the custom HTTP handlers.

For details of the Java packages and classes see the javadoc distributed with the Genero Ghost Client in your FGLGWS installation. The javadoc is found in your `GGCDIR/doc/javadoc/` directory. See the help file by launching the `/doc/javadoc/index.html` file in your browser.
The BDL API implements HTTP handlers through the Java API HTTP handler interface. The option `--http-handler http_handler` is used to specify the handler at the command line.

**Related concepts**

- `ggcadmin` on page 50

The ggcadmin is an administration tool providing commands to start and stop the BDL scenario server, and to run Java scenarios in either TCP direct or UA mode through the GAS.

**GIPSSOHandler**

The Genero IDP Single sign on HTTP handler provides support to authenticate a user.

**Syntax**

```java
com.fourjs.ggc.httphandler.GIPSSOHandler,
auth:username:password
```

It accepts a parameter providing the SSO login details.

- `auth:username:password`

**Usage**

The GIP SSO Handler class is supported by the HTTP Handler interface of the Java API, which must be implemented to provide HTTP handling for SSO.

The BDL API does not support custom HTTP handlers but you can use the Java HTTP handlers in your BDL scenario, by specifying the option `--http-handler http_handler` in the command line.

One or more `--http-handler` options can be used in the command string.

In the example, this option is used to launch the GIP Console application with authentication parameters. (Line breaks have been added to the command examples to improve readability.)

**Example: SSO authentication**

In the example: `auth:admin:admin` instructs the GIPSSOHandler to log in with the `admin` user name and password.

```bash
fglrun consoleapp_login ua
   --url http://myserver:6394/ua/r/admin/ConsoleApp
   --http-handler com.fourjs.ggc.httphandler.GIPSSOHandler,auth:admin:admin
```

**Status codes**

Status codes are used to give GGC status.

Status codes provide status after a test action is performed, to indicate success or errors that can happen when executing a scenario. Table 47: Genero Ghost Client Status codes on page 111 provides a list of the status codes that may be returned. Table 48: Genero Ghost Client Status codes (Protocol and Internal errors) on page 112 describes protocol and internal status errors that may occur.

**Table 47: Genero Ghost Client Status codes**

<table>
<thead>
<tr>
<th>Status</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SUCCESS</td>
<td>Success.</td>
</tr>
<tr>
<td>CHECK_FAILURE</td>
<td>A check generated by the scenario generator failed.</td>
</tr>
<tr>
<td>Status</td>
<td>Description</td>
</tr>
<tr>
<td>------------------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>UI_NOT_FOUND</td>
<td>The UserInterface node was not found.</td>
</tr>
<tr>
<td>WINDOW_NOT_FOUND</td>
<td>No current window found.</td>
</tr>
<tr>
<td>DIALOG_NOT_FOUND</td>
<td>No current dialog found.</td>
</tr>
<tr>
<td>FORM_NOT_FOUND</td>
<td>No form found.</td>
</tr>
<tr>
<td>FIELD_NOT_FOUND</td>
<td>The specified form field was not found in the form.</td>
</tr>
<tr>
<td>FIELD_NOT_VISIBLE</td>
<td>The specified form field is not visible. It can not be focused.</td>
</tr>
<tr>
<td>TABLE_NOT_FOUND</td>
<td>The specified table is not found in the form.</td>
</tr>
<tr>
<td>COLUMN_NOT_FOUND</td>
<td>The specified column is not found in the table.</td>
</tr>
<tr>
<td>ACTION_NOT_FOUND</td>
<td>The specified action is not found in the dialog.</td>
</tr>
<tr>
<td>ACTION_NOT_ACTIVE</td>
<td>The specified action is not active.</td>
</tr>
<tr>
<td>STARTMENU_NOT_FOUND</td>
<td>The specified start menu is not found.</td>
</tr>
<tr>
<td>BAD_PARAMETER</td>
<td>Invalid parameter provided. For example, an invalid column index. A negative value or a value greater than the table's column count will be reported. The error message contains details of the error.</td>
</tr>
<tr>
<td>ILLEGAL_STATE</td>
<td>Occurs when an action is requested but the current state of the DVM does not allow it. Example: Trying to execute an action when the program has ended.</td>
</tr>
</tbody>
</table>

Table 48: Genero Ghost Client Status codes (Protocol and Internal errors)

<table>
<thead>
<tr>
<th>Status</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>PROTOCOL_EXCEPTION</td>
<td>Indicates a failure in the protocol between the client (GGC) and the server / DVM. This error can be raised in following situations:</td>
</tr>
<tr>
<td></td>
<td>• The data received from the server/DVM is invalid.</td>
</tr>
<tr>
<td></td>
<td>• The GGC could not receive data from the server/ DVM. Occurs if the DVM exits prematurely for example.</td>
</tr>
<tr>
<td></td>
<td>• The GGC could not send data to the server/DVM. Occurs if the DVM exits prematurely for example.</td>
</tr>
<tr>
<td></td>
<td>• A child DVM failed to connect within DVM_AVAILABLE timeout.</td>
</tr>
<tr>
<td></td>
<td>• The GGC failed to start the DVM process (TCP/ Direct mode).</td>
</tr>
<tr>
<td></td>
<td>The associated error message provides additional error details.</td>
</tr>
<tr>
<td>INTERNAL_ERROR</td>
<td>Reflects an unexpected internal state. Should not happen.</td>
</tr>
<tr>
<td>NOTIMPLEMENTED</td>
<td>Feature not implemented.</td>
</tr>
</tbody>
</table>
The Genero Ghost Client installation includes sample test scenarios for you to explore in the FGLDIR/testing_utilities/ggc/src/samples directory of your FGLGWS package.

In the samples directory, you find the demo directory containing some Java examples. The SimpleScenario demo is used as a simple test against the price application. See Write a Java test on page 30.

The SimpleScenarioProvider demonstrates how an application starting a child application requires multiple scenarios and how the scenario provider provides a scenario instance when requested.

**Important:** Use Genero BDL to create GGC applications. Genero applications will work for most unit and load tests. Java should only be used to perform critical load testing.

**Explore the scenario provider**

Explore the Genero Ghost Client simple scenario provider demo.

**About this task:**

The SimpleScenarioProvider demonstrates how an application starting a child application requires multiple scenarios and how the scenario provider provides a scenario instance when requested.

This demo provides you with a simple template to use as a reference when you start creating your own test scenarios that need multiple scenarios. In this example, only one scenario (SimpleScenario) is registered and provided to the scenario provider.

1. Compile the test scenarios in the demo package.
   
   From the GGCDIR/src/samples directory, run the java command to compile the SimpleScenarioProvider and the SimpleScenario, which is used as the scenario for the test:
   
   ```
   javac demo/*.java
   ```

2. Execute the test scenario (direct mode):

   **Important:** The option used is --scenario-provider, to specify that a scenario provider class (SimpleScenarioProvider in the example) is being used to run the test.

   - **UNIX® like OS:**
     ```
     ggcadmin tcp -w ${GGCDIR}/src/quick-start -c "fglrun price" --scenario-provider demo.SimpleScenarioProvider
     ```

   - **Windows®:**
     ```
     ggcadmin tcp -w "%GGCDIR%/src/quick-start" -c "fglrun price" --scenario-provider demo.SimpleScenarioProvider
     ```

3. Execute the test scenario (using the Genero Application Server).

   Make sure that the GAS standalone dispatcher httpdispatch is started and that you can access http://localhost:6394/ua/r/price from your browser. Then run the command to test it.

   ```
   ggcadmin ua --url http://localhost:6394/ua/r/price --scenario-provider demo.SimpleScenarioProvider
   ```

4. To execute the test scenario using the Genero Application Server for Java, create an empty war file pointing to the test scenario.

   Refer to the *Genero Application Server for Java User Guide* for details on creating the war file.
Upgrading the GGC

Discover features added with each GGC release and identify any upgrade requirements.

GGC New Features

These topics provide a look back at the new features introduced with each release of the Genero Ghost Client.

GGC 2.00 new features

A summary of new features and changes in functionality introduced with Genero Ghost Client 2.00.

Important: This page covers only those new features introduced with the Genero GGC version specified in the page title. Check prior new features pages if you migrate from an earlier version. Make sure to also read the upgrade guide corresponding to this Genero version.

Corresponding upgrade guide: GGC 2.00 upgrade guide on page 116.

Table 49: What’s new in GGC 2.00

<table>
<thead>
<tr>
<th>Overview</th>
<th>Reference</th>
</tr>
</thead>
<tbody>
<tr>
<td>GGC is now a <strong>centralized backend</strong> to provide better load testing.</td>
<td>See <a href="#">The Genero Ghost Client framework</a> on page 8.</td>
</tr>
<tr>
<td>A method is provided for multi-row selection.</td>
<td>See <a href="#">setRowSelection()</a> on page 85</td>
</tr>
<tr>
<td>Methods are provided to retrieve column values in a matrix, table, tree,</td>
<td>See <a href="#">getColumnValue()</a> on page 83 and <a href="#">getColumnValues()</a> on page 83</td>
</tr>
<tr>
<td>or screen record in your test scenarios and to return values for a</td>
<td></td>
</tr>
<tr>
<td>column at a specified row.</td>
<td></td>
</tr>
<tr>
<td>Network load is simulated by loading all images attached to the AUI Tree.</td>
<td>No additional reference.</td>
</tr>
<tr>
<td>(Not done in TCP/Direct mode, only via GAS)</td>
<td></td>
</tr>
<tr>
<td>The GGC is embedded in the FGLGWS package in the $FGLDIR/</td>
<td>See <a href="#">Install and configure for Genero Ghost Client</a> on page 8</td>
</tr>
<tr>
<td>testing_utilities/ggc directory.</td>
<td>No additional reference.</td>
</tr>
<tr>
<td>Documentation about the APIs that make up the GGC framework can be</td>
<td></td>
</tr>
<tr>
<td>accessed:</td>
<td></td>
</tr>
<tr>
<td>• For details of the Java packages and classes see the javadoc distributed</td>
<td></td>
</tr>
<tr>
<td>with the Genero Ghost Client in your FGLGWS installation. The javadoc</td>
<td></td>
</tr>
<tr>
<td>is found in your GGCDIR/doc/javadoc/ directory. See the help file by</td>
<td></td>
</tr>
<tr>
<td>launching the /doc/javadoc/index.html file in your browser.</td>
<td></td>
</tr>
<tr>
<td>• For details of the BDL API, see <a href="#">Genero BDL API for GGC</a> on page 63</td>
<td></td>
</tr>
<tr>
<td>Templates with snippets of generic code for testing are available in the</td>
<td>See <a href="#">Templates</a> on page 13</td>
</tr>
<tr>
<td>installation directory. They are located in GGCDIR/template.</td>
<td></td>
</tr>
<tr>
<td>A feature of templates allows you to create a skeleton scenario with code</td>
<td></td>
</tr>
<tr>
<td>from the header and footer templates. This provides the code that forms</td>
<td></td>
</tr>
<tr>
<td>the basis of the test application.</td>
<td>See <a href="#">Write a Genero BDL test</a> on page 26</td>
</tr>
<tr>
<td></td>
<td>See <a href="#">Write a Java test</a> on page 30</td>
</tr>
</tbody>
</table>
Table 50: Engine and Architecture

<table>
<thead>
<tr>
<th>Overview</th>
<th>Reference</th>
</tr>
</thead>
<tbody>
<tr>
<td>The <strong>ggcadmin</strong> is an administration tool providing commands to start and stop the BDL scenario server, and to run Java scenarios in either direct or UA mode.</td>
<td>See <strong>ggcadmin</strong> on page 50</td>
</tr>
<tr>
<td>The <strong>ggcgen</strong> is the tool that you use to generate test scenarios. There are commands to generate BDL and Java scenarios.</td>
<td>See <strong>ggcgen</strong> on page 55</td>
</tr>
<tr>
<td>The logging mechanism for the Genero Ghost Client allows you to control the level of logs and the output messages.</td>
<td>See <strong>log.properties</strong> on page 61</td>
</tr>
<tr>
<td>To ease Single Sign on detection, an HTTP handler named <strong>GIPSSOHandler</strong> is available.</td>
<td>See <strong>GIPSSOHandler</strong> on page 111</td>
</tr>
<tr>
<td>Support for front calls is available.</td>
<td>See Front calls on page 43 and Default front calls on page 43.</td>
</tr>
<tr>
<td>GGC tools (<strong>ggcadmin</strong>, <strong>ggcgen</strong> and BDL scenarios) can load command options from a default argument file.</td>
<td>See Default argument file on page 59</td>
</tr>
</tbody>
</table>

Table 51: Test generation

<table>
<thead>
<tr>
<th>Overview</th>
<th>Reference</th>
</tr>
</thead>
<tbody>
<tr>
<td>There is a debugging tool that is activated by the &quot;**--dump_*&quot; options of the <strong>ggcgen</strong> tool.</td>
<td>See <strong>ggcgen</strong> on page 55</td>
</tr>
<tr>
<td>The <strong>ggcgen --no-wait</strong> option allows for the generation of test scenarios without instructions for delays between actions.</td>
<td>See <strong>ggcgen</strong> on page 55 and Test speed ratio on page 41</td>
</tr>
<tr>
<td>The <strong>--check-messages</strong> option of the <strong>ggcgen</strong> tool generates code that tests <strong>MESSAGE</strong> and <strong>ERROR</strong> messages. Templates are provided for these checks.</td>
<td>See <strong>ggcgen</strong> on page 55</td>
</tr>
</tbody>
</table>

Table 52: Running tests

<table>
<thead>
<tr>
<th>Overview</th>
<th>Reference</th>
</tr>
</thead>
<tbody>
<tr>
<td>The <strong>--environment-file</strong> option of the <strong>ggcadmin</strong> tool (Java) and the GGC scenario program (BDL) allows environment variables to be provide in a file for the environment of the DVM that is started by the scenario.</td>
<td>See <strong>ggcadmin</strong> on page 50 or GGC scenario program options on page 57 and Set test environment on page 40.</td>
</tr>
<tr>
<td>The <strong>--fename client-name</strong> option of the <strong>ggcadmin</strong> tool (Java) and the GGC scenario program (BDL) allows you to specify an alternate default client name.</td>
<td>See <strong>ggcadmin</strong> on page 50 or GGC scenario program options on page 57.</td>
</tr>
</tbody>
</table>

GGC 1.10 new features

A summary of new features and changes in functionality introduced with Genero Ghost Client 1.10.

The following changes and enhancements are relevant to this publication.

Table 53: What's new in GGC 1.10

<table>
<thead>
<tr>
<th>Overview</th>
<th>Reference</th>
</tr>
</thead>
<tbody>
<tr>
<td>The Genero Ghost Generator now ships as part of the FGLGWS package.</td>
<td>See Install and configure for Genero Ghost Client on page 8.</td>
</tr>
</tbody>
</table>
Upgrade Guides for Genero Ghost Client

These topics provide a look back at the new features introduced with each release of the Genero Ghost Client.

**Important:** Each upgrade guide is an incremental upgrade guide that covers only topics related to a specific version of Genero Ghost Client. It is important that you read all of the upgrade guides that sit between your existing version and the desired version.

### GGC 2.00 upgrade guide

These topics describe product changes you must be aware of when upgrading to version 2.00.

**Important:** This is an incremental upgrade guide that covers only topics related to the Genero Ghost Client version specified in the page title. Check prior upgrade guides if you migrate from an earlier version. If you are migrating your test scenarios, make sure you compile with this version. Make sure to also read about the new features for this version.

Corresponding new features page: [GGC 2.00 new features](#) on page 5.

### Migrating from GGC Version 1 to 2

Genero Ghost Client version 2 is not compatible with version 1. If your tests were written or recorded for version 1, you must use the GGC version 1 to run them until you are ready to migrate to version 2.

If you are migrating your test scenario generated with version 1, you will need to rewrite test code or re-record logs and generate test scenarios using GGC version 2. For more information see the [Summary of changes from version 1 to 2](#) on page 117.

### Deprecated ggcadmin uastart and tcpstart commands

The `ggcadmin` on page 50 `uastart/tcpstart` commands are deprecated. You will need to replace these commands with the `ggcadmin ua/tcp`. The `uastart` and `tcpstart` commands continue to be supported but will be removed in a future release.

### Deprecated ggcadmin ua --start-url parameter

The `ggcadmin ua --start-url` parameter is deprecated. You will need to replace this with `--url` in commands used to specify the URL to start the application for testing. For example,

```
ggcadmin ua --url http://localhost:6394/ua/r/myApp --scenario myTests.MyScenario
```
The `--start-url` parameter continues to be supported but will be removed in a future release.

**Deprecated ggcadmin tcp/ua short options -fg and -sr**

To support use of standard `getopt` (UNIX® like OS) short options with two characters have been renamed with one character. This effects `ggcadmin` on page 50 `tcp` and `ggcadmin` `ua` options `-fg` and `-sr`, which are now deprecated. You will need to replace these in your `ggcadmin` `tcp/ua` commands. For example:

- In a command to specify a GUI to forward to, replace `-fg` with `-f`

  ```
  ggcadmin tcp -w path/to/myapp -f localhost:0 -c "fglrun myapp" --scenario test_dir.myapp_test
  ```

- In a command to specify the speed ratio of a test, replace `-sr` with `-s`

  ```
  ggcadmin tcp -w path/to/myapp -s 0.1 -c "fglrun myapp" --scenario test_dir.myapp_test
  ```

**Front end defaults change**

Starting with GGC 2.00.13, the default client name returned by the GGC changes. Prior to this release, the default returned was always "GGC". Starting with this release:

- When using an HTTP connection (via the Genero Application Server (/ua/)), GGC uses "GBC".
- When using a direct connection (TCP/IP) or a local connection, GGC uses "GDC".

**Deprecated status codes**

The Status Code `FIELD_NOT_ACTIVE` is no longer supported. If your tests were written using this code, you will need to rewrite your test code.

**Summary of changes from version 1 to 2**

**Setting the environment**

CLASSPATH for Java is set by running the `envcomp` environment script. See Install and configure for Genero Ghost Client on page 8.

**Generating a scenario**

Scenarios are generated from a `guiilog`, or a GDC log in v2 using the `ggcgen` on page 55 tool. See Recording logs and generating scenarios on page 33.

**Framework changes:**

GGC v2 encapsulates and simplifies the underlying Java framework. BDL scenarios are no longer instantiate directly through the Java Bridge. Now they connect to a GGC backend using channels. See The Genero Ghost Client framework on page 8.

- The `GhostGenerator` interface no longer exists, its function is replaced with the `ggcgen` command.
- The `Launcher` interface function is replaced with the functionality of `ggcadmin` on page 50 or `fglrun` commands `tcp/ua`. You can provide a list of scenarios to test in a parameter.

Thread delays are specified with the `--speed-ratio` option of the `ggcadmin` (for Java tests) or the `fglrun` command (for BDL tests). See Test speed ratio on page 41.
If more than one scenario is to be tested, a scenario provider (ScenarioProvider) interface is created and populated with the scenarios. A configuration (TCP/UA) is created, followed by a SessionManager, and a new session is added using the previously built configuration and scenario provider.

The created session (UA or TCP session) deals with the main scenario and, if some child VMs are started, with the corresponding child scenarios. See Explore the scenario provider on page 113.

**Execution statistics**

The output when a scenario ends differs in GGV v2. Summary statistics about the execution are displayed when tests end. When test checks or scenarios fail, these are displayed in the summary at the end. See Quick start with Ghost Client on page 17 and Session information on page 98.

**Templates**

Templates on page 13 are new to v2. These correspond to checks that existed in v1, which were enabled by default.

Now options are provided so you can control which checks will be generated. You can also control how they are generated by updating the templates if needed. See Customizing a template on page 15.

A feature of templates allows you to create a skeleton scenario for an empty test using the header and footer templates.

**Logging mechanism**

The logging mechanism is different and is configured with the log.properties on page 61 file. GHOSTLOG of v1 is not used in v2.

**Debugging**

If you activate the "--dump_*" options feature of the ggcgen on page 55 tool, it in-lines information as comments about the state of the AUI tree at the moment an action is executed. For instance, a full tree dump is recorded in the scenario log file.

This is a debugging tool, useful for finding where a scenario execution differs from a recorded session. You may find it useful to turn on this feature particularly for support requests.

**Macros**

The use of macros in GGC version 1 has been replaced by Templates on page 13.

**Front calls**

Support for front calls has been enhanced in v2. See Front calls on page 43.
Copyrights and Trademarks

The Copyright and trademark page is shared amongst all Four Js documentation. Appearance in this page does not guarantee a mention within the current document.

Copyright notice
© Copyright Four Js Development Tools, Inc. Use, duplication or disclosure restricted.

Trademarks
Adobe, the Adobe logo, PostScript, and the PostScript logo are either registered trademarks or trademarks of Adobe Systems Incorporated.

IBM and WebSphere are trademarks or registered trademarks of International Business Machines Corp., in many jurisdictions worldwide. Other product and service names might be trademarks of IBM or other companies. A current list of IBM trademarks is available on the Web at Copyright and trademark information.

Linux is a registered trademark of Linus Torvalds.

Microsoft, Windows, Windows NT, and the Windows logo are trademarks of Microsoft Corporation.

Oracle and Java are registered trademarks of Oracle and/or its affiliates. Other names may be trademarks of their respective owners.

DEBIAN is a registered trademark owned by Software in the Public Interest, Inc. Debian trademark is a registered United States trademark of Software in the Public Interest, Inc., managed by the Debian project. Genero is not affiliated with Debian. Debian is a registered trademark owned by Software in the Public Interest, Inc.

Red Hat is a registered trademark of Red Hat, Inc. in the United States and other countries.

Red Hat Enterprise Linux is a registered trademark of Red Hat, Inc. in the United States and other countries.

Other company, product, or service names may be trademarks or service marks of others.
Index

B

BDL

api

action 78
collapseTree 87
expandTree 87
FrontCallAnswer
functionNotFound 106
moduleNotFound 107
notProcessed 107
returnString 108, 108
stackError 109
success 109
userError 110
FrontCallAnswer type 75
FrontCallRequest
getFunctionName 103
getModuleName 103
getParameterCount 104
getParameterValue 104
getReturnCount 105
FrontCallRequest type 75
getApplicationName 99
getCount 99
columnCount 96
columnName 97
columnValue 83
columnValues 83
getCurrentColumn 97
currentRow 97
gMessage 94
gState 100
gState 100, 101
gStatistics 100, 101
gTableOffset 98
gTableSize 98
getValue 91
gValues 92
gWidgetItem 92
gWindowName 92
gWindowTitle 93
hideTableColumn 84
isActionActive 95
key 78
message type 76
notifyCheckFailure 101
setCellFocus 84
setFieldValue 79
setFocus 79
setRowFocus 84
setRowSelection 85

setTableOffset 85
setTableSize 86
setValue 80
showTableColumn 86
sortTable 86
statistics type 76
API

action and key functions 78
action state functions 95
field interaction functions 79
front call requests 102, 105
MESSAGE and ERROR functions 93
retrieving data functions 89
session information functions 98
table information functions 96
table interaction functions 81
tree view interaction functions 87
types 73
API functions 63
BDL scenario program options 57

C

customizing a template 15
demo
SimpleScenarioProvider 113
demos 113

evironment settings 40

F

front call handlers 43
front calls
implement in test 44
manage in test 43

G

generate test from GBC log 40
generate test from GDC log 39
generate test sets from logs 39
Genero BDL test 26
Genero Ghost Client
directory 12, 13
template examples 14
Genero Java test 30
Ghost Client
default argument file 59
get version 60
log files 62
log.properties 61
logging 61
options 49
quick start guides 17
scenario 25, 26
testing 25
Ghost Client Testing Tool 7

H

how it works 8
HTTP handler
  GIPSSOHandler 111
HTTP handler API 110

I

infrastructure 8
Installation 8

J

Java
  API functions 110

N

new features
  version information 114

R

record logs, generate scenarios 33
record scenario logs 34
run BDL test 28
run Java test 32

S

speed-ratio 41
status codes 111

T

testing
  certificate checks 42
  single sign on 43
  unit tests 26

U

upgrade guide
  Version 2.00 116
Upgrading 114

W

what's new
  upgrade guides 116
  Version 1.10 115
  Version 2.00 5, 114
  version list 114