

2.11 - Operational Test Data with DPLs

Operational Test Data with DPLs — Appvance IQ documentation.

- [Operational Test Data with DPLs in AIQ](#)
- [AIQ DPL Types](#)
- [Configure AIQ DPLs](#)
- [Providing Data to Distributed Tests](#)
- [Configure Dynamic DPL Data](#)
- [Array Generator DPL \(Test Scenario, Functional Test\)](#)
- [WhiteBoard DPL Usage in AIQ](#)
- [Use Scenario-Wide DPL Data](#)
- [Use Hash DPL Data in Scenario Editor Tests](#)
- [Configure JavaScript DPL Data](#)
- [Use DPL Data Sources in AIQ](#)
- [Communication Intra-Steps Using TheLocker](#)
- [The RSC Resource](#)
- [Creating Your Own DPL Using JavaScript in AIQ.](#)
- [Specific Types of DPLs](#)
- [Use DPL Data in Scenario Editor Tests](#)
- [Synthetic DPL Data in AIQ](#)
- [Encrypt/Decrypt DPL](#)
- [Operational Test Data FAQ](#)

Operational Test Data with DPLs in AIQ

Recommended guidance

Use DPLs to manage operational test data for Scenario Editor and Service Suite assets when tests need repeatable, controlled, or generated values. Treat DPLs as part of the test design, not as incidental data pasted into individual steps.

Planning checklist

- Define which steps consume each data value.
- Choose static, CSV/file-backed, synthetic, dynamic, or service-backed data based on the test objective.
- Document reset, cleanup, ownership, and environment-specific differences.
- Avoid unsecured secrets, production customer data, and values that cannot be reproduced.

Validation workflow

1. Run a small data sample before scaling execution.
2. Confirm each row or generated value maps to the expected scenario behavior.
3. Review failures for data exhaustion, malformed values, duplicates, or environment mismatch.
4. Keep DPL changes under review when they affect regression results.

Last updated: October 6, 2026

AIQ DPL Types

Recommended guidance

DPLs provide operational test data for AIQ scenarios and services. Choose the DPL type based on data ownership, repeatability, volume, security, and whether values are static, generated, or retrieved from an external system.

Common DPL considerations

- Static data for repeatable regression tests.
- CSV or file-backed data for controlled row-based execution.
- Dynamic or generated data for unique values per run.
- Service-backed data when the test needs values from an API or system of record.
- Secure handling for credentials, personal data, or customer-sensitive values.

Selection checklist

1. Define whether the scenario needs unique, reusable, random, or environment-specific data.
2. Confirm how data is reset or cleaned up after execution.
3. Document which scenario steps consume each DPL value.
4. Do not store secrets or production customer data in unsecured DPL sources.

Last updated: October 6, 2026

Configure AIQ DPLs

Recommended guidance

Configure AIQ DPLs so scenarios and services receive the right data at the right time. DPL configuration should define source, format, ownership, lifecycle, reset behavior, and security boundaries.

Configuration checklist

- Choose the correct DPL type for static, file-backed, generated, dynamic, or service-backed data.
- Confirm field names, formats, row counts, uniqueness, and required values.
- Define how the test consumes rows and what happens when data is exhausted.
- Document cleanup, reset, and environment-specific overrides.
- Avoid storing passwords, tokens, or production customer data in unsecured DPL sources.

Validation workflow

1. Run a small scenario with known data.
2. Check Output Logs for missing or malformed values.
3. Verify assertions confirm the data was used correctly.
4. Review DPL changes before large regression or performance runs.

Last updated: October 6, 2026

Providing Data to Distributed Tests

AIQ is a distributed test system. Multiple virtual users run in a TestNode and tests run across one or more TestNodes. Tests often need unique data for each virtual user. For example, a test of a banking system needs a separate and unique customer number for each virtual user running in the test. AIQ offers 3 solutions to meet your testing needs.

Common Test Data, No Coordination

AIQ's CSV DPL sends a copy of a single CSV file to each TestNode running a test. Each virtual user uses data in the CSV file. Within a TestNode each user shares the CSV file and shares a single pointer to the row of data in the CSV file.

CSV files are distributed to each TestNode

To use this configuration, use the Appvance Scenario Builder to configure a Test Use Case that uses a CSV DPL /Hash DPL. Set the DPL Action setting to Get Next Row of Data. When you run the test AIQ will send a copy of the defined CSV file to each TestNode. AIQ creates a single DPL for all virtual users in the TestNode. The DPL has a raw pointer. The pointer is initially set to the first row of data. The DPL reads the first row of data and injects the data fields into the test script. The DPL moves the pointer to the next row of data. At the last row of data, the DPL moves the pointer to the first row of data.

Unique Provided Data

Many times the test architecture requires each virtual user to use unique data. For example, when testing a banking system each virtual user needs to log in to the banking application using its own user-id and password. AIQ provides the Unique DPL to provide each virtual user's DPL with its own data. The data comes from a single CSV file. Each virtual user uses its local DPL to communicate over a TCP/IP request to a single Unique DPL service. The Unique DPL reads from a single CSV file.

Unique data is distributed to each virtual user

The communication overhead for Unique DPL is minimal. Appvance staged tests up to 100,000 virtual users and observed less than 1% network bandwidth overhead and less than 20-millisecond wait times when using UniqueDPL.

Running a Unique DPL scenario using a CSV file which has duplicate rows

Consider the following scenario to understand the unique DPL functionality.

Suppose the user is creating an Appvance Designer script with DPL enabled. The created DPL contains duplicate values for the field that was data enabled and considers the scenario for 2 executions and 2 users and it should have multiple test Nodes. The Unique DPL works in such a way that it will not validate whether the CSV file has repeated rows or not, that is, it does not check for the content of the rows. In AIQ, the Unique DPL functionality is that it will make sure that among all running users in all running TestNodes in a single execution, each row of the CSV data file will be used only once until all rows available in the DPL are used.

Note: Please make sure the CSV file has unique data rather than duplicated rows.

Transactional Data Provided To Each Virtual User

AIQ's RDBMS DPL enables tests to receive operational test data from a relational database. For example, configure a MySQL instance to serve transactional data to the virtual users. One Appvance customer stored invoice data for a test of an Enterprise Resource Planning (ERP) system. A stored procedure (StoredProc) in MySQL provides invoice data to the virtual users, including part numbers, part costs, quantities, and shipping details. The StoredProc enables each virtual user to receive its own unique data from the common data store.

Transactional data is distributed to each virtual user

The above example showed how to use data from a CSV file. DPLs work in the same way with data from a Relational Database (RDBMS) and other data-generating objects. See the DPL Recipes section for details.

If you have any feedback on this topic, please contact us at documentation@appvance.ai

Configure Dynamic DPL Data

Configure Dynamic DPL Data

Use a Dynamic DPL when a Scenario Editor test needs generated values at runtime, such as random numbers, generated strings, timestamps, or values within a defined range. Dynamic DPLs are useful when the test data does not need to come from a fixed file or database source.

When to use it

- Generate unique values during each execution.
- Avoid hard-coded test data in Scenario Editor steps.
- Keep data rules in one reusable DPL instead of duplicating logic across scenarios.

Implementation notes

1. Define the dynamic data rule in the DPL configuration.
2. Map the DPL output to the Scenario Editor input that needs the generated value.
3. Run a short validation execution and confirm the generated values appear in the output or reports.

Keep generated values within realistic ranges for the application under test. If a test requires a stable, repeatable data set, use a static source DPL instead.

Last updated: September 18, 2026

Array Generator DPL (Test Scenario, Functional Test)

To run Java code with array arguments in Appvance, use the **Array Generator DPL**. To execute Java code that has a `main` method in Scenario Builder, select the **Array Generator DPL** type and `String[]` from the **Actions** dropdown. Add the argument name and pass a value by clicking the **Add Val** link.

The main method receives the array from the generator.

With the **Array Generator DPL**, you can send an array of values as a single parameter containing multiple entries, for example a String array:

```
[ "string1", "string2", "string3" ]
```

This allows you to provide an input parameter for a method that requires an array-like parameter:

```
public void myMethod(String[] args) {  
}
```

You can also use Object arrays. Example:

```
[  
  { name: 'Honda', models: [  
    { name: 'Accord', features: ['2dr', '4dr'] },  
    { name: 'CRV', features: ['2dr', 'Hatchback'] },  
    { name: 'Pilot', features: ['base', 'superDuper'] }  
  ] },  
  { name: 'Toyota', models: [  
    { name: 'Prius', features: ['green', 'superGreen'] },  
    { name: 'Camry', features: ['sporty', 'square'] },  
    { name: 'Corolla', features: ['cheap', 'superFly'] }  
  ] }  
]
```

Types available for arrays: `Object`, `String`, `Integer`, `int`, `Byte`, `byte`, `Double`, `double`, `Short`, `short`, `Float`, `float`, `Character`, `Boolean`.

Sample Java Code with an int Array

```
package com.package1;

import java.util.Arrays;

public class Array {
    public static void main(String[] args) throws Exception {
    }

    public void printArray(String message, int array[]) {
        Arrays.sort(array);
        System.out.println(message + ": [length: " + array.length + "]");
        for (int i = 0; i < array.length; i++) {
            if (i != 0) {
                System.out.print(", ");
            }
            System.out.print(array[i]);
        }
        System.out.println();
        int index = Arrays.binarySearch(array, 2);
        System.out.println("Found 2 @ " + index);
    }
}
```

Create a **.jar** file from the Java code above. Go to Scenario Builder and select the **Functional Test** test type. Add a test case using the **Java Test** type and browse to the **.jar** file you created. The code above takes two arguments: the first is a **String**, and the second is an **int** array. Pass the arguments in the same order in Appvance so the scenario runs successfully.

Two arguments using array generator

Set the log level to 100% in the Options panel. Save and play the scenario in a Controller window.

Output:

```
Two arguments using array generator
```

If you have any feedback on this topic, please contact us at documentation@appvance.ai

WhiteBoard DPL Usage in AIQ

Last reviewed: September 29, 2026

Use this article when test data must be shared or manipulated through a WhiteBoard DPL workflow.

What changed in this article

- Updated the title casing and tied the article to Operational Test Data/DPL usage.
- Added guidance on when WhiteBoard DPL is appropriate versus CSV, Dynamic DPL, or JavaScript DPL.
- Added validation checks for generated rows, shared state, cleanup, and repeatability across iterations.

Validation workflow

1. Confirm the test needs WhiteBoard-style shared or generated data rather than a static CSV.
2. Validate produced values across multiple iterations and users.
3. Confirm cleanup or reset behavior before using the DPL in a repeatable test run.

Review note: Clarified the DPL use case so WhiteBoard DPL is not chosen when a simpler data source would work.

Last updated: September 29, 2026

If you have feedback on this topic, contact documentation@appvance.ai.

Use Scenario-Wide DPL Data

Last reviewed: September 30, 2026

Use this article when a Scenario Editor test needs one DPL data source available across multiple steps or actions in the same scenario.

What changed in this article

- Normalized the title and explained scenario-wide DPL scope in current Scenario Editor terms.
- Added guidance for choosing scenario-wide DPLs versus step-specific data when values must be reused across the run.
- Added checks for row selection, iteration behavior, missing values, concurrency, and sensitive data handling.

Validation workflow

1. Choose scenario-wide DPL only when multiple steps need the same data context.
2. Validate row selection and iteration behavior with a small run before scaling.
3. Confirm logs and reports expose enough row context for debugging without exposing sensitive values.

Review note: This page now differentiates shared scenario data from one-off step data.

Last updated: September 30, 2026

If you have feedback on this topic, contact documentation@appvance.ai.

Use Hash DPL Data in Scenario Editor Tests

Last reviewed: September 30, 2026

Use this article when a Scenario Editor test needs key-value style test data supplied through a Hash DPL.

What changed in this article

- Replaced terse Hash DPL wording with Scenario Editor-focused data guidance.
- Clarified when a Hash DPL is appropriate: lookup tables, keyed records, expected values, and reusable test configuration.
- Added validation checks for key uniqueness, missing keys, data refresh, sensitive values, and deterministic replay.

Validation workflow

1. Define the key field and confirm every required test case has a matching value.
2. Reference the Hash DPL from the Scenario Editor test and validate both a matching-key and a missing-key case.
3. Keep sensitive values out of plain test data or mask them before logging results.

Review note: This update keeps Hash DPL guidance practical without turning it into a generic data-storage page.

Last updated: September 30, 2026

If you have feedback on this topic, contact documentation@appvance.ai.

Configure JavaScript DPL Data

Configure JavaScript DPL Data

Use a JavaScript DPL when test data needs lightweight transformation before it is consumed by a Scenario Editor test. Keep the script focused on data preparation; avoid placing test flow, browser automation, credentials, or environment-specific paths in the DPL.

Recommended use

- Normalize or format values before sending them to the application under test.
- Combine fields from an approved data source into a value required by a step.
- Generate simple derived values that are safe to calculate at runtime.

Validation checklist

1. Confirm the JavaScript returns the expected value for a small sample.
2. Handle empty or malformed input values predictably.
3. Verify the DPL mapping in a short Scenario Editor execution before scaling the test.

Do not embed secrets in the DPL script. Store credentials and environment-specific settings in approved configuration locations.

Last updated: September 18, 2026

Use DPL Data Sources in AIQ

Use DPL Data Sources in AIQ

DPLs provide reusable data to AIQ tests. They keep test data and data-generation rules separate from Scenario Editor steps so scenarios remain easier to maintain.

Basic workflow

1. Create or select the DPL that represents the required data source or generated data rule.
2. Map the needed DPL fields to the Scenario Editor inputs.
3. Choose the execution behavior, such as using the next available row or reusing a value.
4. Run a short validation execution and review the output before using the DPL in a larger run.

Good practices

- Use clear field names that match the business meaning of the data.
- Keep credentials and environment-specific values out of the DPL article and scenario content.
- Document data assumptions, such as required formats, uniqueness, and reset behavior.

Last updated: September 18, 2026

Communication Intra-Steps Using TheLocker

Appvance IQ's architecture allows you to integrate different testing tools and technologies into one single orchestration framework. Each one of these tools and technologies works independently, leaving some gaps that won't let them work as a team with other tools. One gap Appvance IQ fills is the external data-driven testing, achieved with the DPLs. The other gap is the actual communication between tools. That is achieved with a special DPL called TheLocker.

By default, all DPLs are read-only. That means you can use them to obtain dynamic data, but not to push data to them. When orchestrating, one test step (performed using one tool, let's say soapUI) may generate data that can be used as input to the next step (say a verification on a web page using Selenium). To communicate these two steps, one solution will be to have one DPL that allows step 1 to push its results into a named "space" where step 2 can read it later on. This is what TheLocker does.

TheLocker metaphor is a simple one. Any tool or step can request a locker space, name it and store any kind of data in it. The subsequent steps are able to retrieve that data using the name of the space. An example may help to understand the concept better.

The first step, as with all the DPLs, is to create an instance of TheLocker:

```
<DataSources><dpl name="myLocker" type="TheLocker"/></DataSources>
```

As special as TheLocker is, it behaves a little differently than regular DPLs. One difference is that a normal DPL instance shares all its data with all the user threads, thus one user changing its data-position will change the position for all the other threads. TheLocker has one copy of the instance for each user, so each one may add different values to the same space and this will not override their partner's data. Another difference is TheLocker always has one instance, no matter how many you define, it will be the same.

Now, to push data. To have data pushed into TheLocker, the testing tool in turn needs to offer a scripting facility. In the example mentioned above, soapUI is the tool that may push data into a namespace and this is done using the groovy scripting facility of soapUI. This is done by creating a Groovy step in soapUI and issuing some code similar to this:

```
def myLocker = new com.Appvance.tool.dpl.provided.TheLocker();
myLocker.put("path1",firstXPath)
myLocker.put("path2",secondXPath)
```

Let's see the selenese commands inside our selenese file, to see how it will pass the data:

```
<tr><td>clickAndWait</td><td> path1</td><td></td></tr><tr><td>clickAndWait</td><td> path2</td><td></td></tr>
```

When running Selenese, the passed data from TheLocker will match the name of the pairs (path1 and path2) and replace those in the Selenese file with the data soapUI placed in TheLocker.

Note this is highly intuitive. Appvance IQ intelligently matches up and attends to the intention of the user to place the values in the correct place.

But the user may have even greater control of TheLocker in a scripting environment. For instance, if the receiving step is a Jython step, you may not need to pass the DPL values as above, you can simply request TheLocker instance and get data from it, like this:

```
from com.Appvance.tool.dpl.provided import TheLocker
...
myLocker = TheLocker()
lockerValues = myLocker .getData();
```

Last updated: June 02, 2023

If you have any feedback on this topic, please contact us at documentation@appvance.ai

The RSC Resource

When working with distributed testing nodes, tests may require additional resources such as CSV files, data files, or other external files. These resources must be transferred to the remote testing node before the test executes.

Because the resource may be stored at a different location on the remote node, a path that was hardcoded when the test was created may no longer be valid. This can cause the test to fail because the required file cannot be found.

Appvance IQ provides the **RSC DPL** to solve this problem. Resources can be declared in the test's `<resources>` section, and the RSC DPL can then be used to retrieve the actual path of a resource at runtime, regardless of where the resource is stored on the testing node.

Declaring Resources

Resources required by the test, including resources used by other DPLs, should be declared in the `<resources>` section.

For example, suppose a `HashDPL` requires a CSV file. The CSV file can exist anywhere on the local machine, but it must be included in the resources section so that Appvance IQ can transfer it to the remote testing node.

```
<resources>
  <data path="c:\mydata\data.csv"/>
</resources>
```

The `<data>` element represents a file resource.

Resources are assigned an index based on their order in the `<resources>` section. The first resource has index **0**.

For example:

```
<resources>
  <data path="c:\mydata\data.csv"/>
  <data path="c:\mydata\users.csv"/>
</resources>
```

In this example:

- `data.csv` is resource **0**
- `users.csv` is resource **1**

Using RSC with another DPL

Once a resource has been declared, the **RSC DPL** can be used to obtain its actual runtime path.

For example, if a `HashDPL` requires the path to a CSV file:

```
<DataSources>
  <dpl name="mydpl" type="HashDPL">
    <argument name="file"
      dpl="rsc"
      value="getDataByIndex"
      index="0"/>
  </dpl>
</DataSources>
```

Here, the `file` argument of the `HashDPL` receives its value from the RSC DPL.

The following attributes are used:

Attribute	Description
<code>dpl="rsc"</code>	Specifies that the RSC DPL is being used.
<code>value="getDataByIndex"</code>	Specifies that the resource path should be retrieved by its index.
<code>index="0"</code>	Identifies the resource declared at index 0 in the <code><resources></code> section.

The value returned by RSC is the **actual path of the resource on the testing node**. This means the test does not need to know or depend on the path used by the remote node.

Using RSC in Test Steps

RSC can also be used directly in test step definitions.

For example, a Jython test step may require the path of a resource:

```
<run name="CheckInventory"
  module="JythonModule"
  testclass="testgen4web"
  method="run"
  langtype="jython">
  <argument name="frozenfile"
    dpl="rsc"
    value="getDataByIndex"
    index="0"/>
  <argument name="ownerinfo"
    dpl="values"
    value="getData"/>

</run>
```

In this example, the `frozenfile` argument receives the runtime path of resource **0**.

The test step can therefore access the file without knowing where Appvance IQ stored the resource on the remote testing node.

Recommended Usage

Use the following approach when a test depends on an external file:

1. Add the file to the `<resources>` section.
2. Determine its resource index based on its position in the section. Remember that indexing starts at **0**.
3. Use the RSC DPL with `getDataByIndex` whenever the file path is required.
4. Avoid hardcoding paths to resources that will be used on distributed testing nodes.

This approach allows the same test to run reliably across different testing nodes and environments without requiring changes to file paths.

Summary

The RSC DPL provides a runtime-safe way to access resources transferred to distributed testing nodes. Instead of relying on a hardcoded local or remote path, the test references the resource by its index in the `<resources>` section.

For example:

```
<resources>
  <data path="c:\mydata\data.csv"/>
</resources>
```

The resource can then be referenced anywhere a file path is required:

```
<argument name="file"
  dpl="rsc"
  value="getDataByIndex"
  index="0"/>
```

The RSC DPL resolves the resource to its actual location on the testing node at runtime.

Creating Your Own DPL Using JavaScript in AIQ.

Creating your own DPL requires Java programming skills and a familiarity with the DPL interface definition. It involves a two-step process: first, you need to implement the DPL main class; second, you need to define a new DPL type for Appvance IQ.

REMEMBER: This is only one of the many ways of creating DPLs (simulated data that can be used for Scripts, Scenarios, and more). If you want to learn more about how to create DPLs and integrate them, visit our LMS and watch these videos:

- <https://academy.appvance.ai/mod/resource/view.php?id=211>
- <https://academy.appvance.ai/mod/resource/view.php?id=206>

Also, if you are interested in reading external data (Excel, CSV, and others):

- <https://academy.appvance.ai/mod/resource/view.php?id=206>

The DPL Java class implements the standard DPL methods including getNextData, getDataByIndex, and more. It needs to implement the com.Appvance.tool.dpl.DataProduction interface. Here is an example:

```
import com.Appvance.tool.dpl.DataProduction;

public class SimpleDPL implements DataProduction {

    public SimpleDPL() { }

    public void setup(String data) {
        System.out.println("This is a setup using data "+data);
    }

    public Object getData() { return null; }

    public Object getRandomData() { return "RANDOM DATA"; }

    public Object getNextData() { return "Simple data: getNextData"; }

    public Object getDataByIndex(int index) {
        return "Simple data: get data by index "+index;
    }

    public void tearDown() { }
```

```
}
```

Once you define and compile a custom DPL packaged into a .jar file, you should use the `<DataSources>` in a TScenario document to identify your new DPL type.

If you have any feedback on this topic, please contact us at documentation@appvance.ai

Specific Types of DPLs

Use DPL Data in Scenario Editor Tests

Use DPL Data in Scenario Editor Tests

Use DPL data sources to provide repeatable test data to Scenario Editor executions. Store source files and database connection details in approved locations and avoid embedding credentials in scripts.

1. Create or select the required data source.
2. Map the needed fields to the relevant test inputs.
3. Configure execution behavior, such as advancing to the next available data row.
4. Run a small validation execution before scaling the test.

Refer to the release-specific DPL documentation for supported source types and configuration fields.

Last updated: September 14, 2026

Synthetic DPL Data in AIQ

Recommended guidance

Use Synthetic DPL data when AIQ tests need generated values that are safe, repeatable enough for validation, and independent from production customer records. Synthetic data should match the format and constraints of the application under test.

Good use cases

- Generating names, IDs, addresses, emails, or other non-sensitive values.
- Creating unique values per run to avoid duplicate-data conflicts.
- Testing validation rules without exposing real customer data.
- Supporting performance or regression runs that need large data volume.

Quality checks

1. Confirm generated values match field length, format, and business rules.
2. Document randomness, uniqueness, and cleanup expectations.
3. Do not use synthetic values that accidentally resemble real sensitive data.
4. Validate failures against both scenario logic and generated data quality.

Last updated: October 6, 2026

Encrypt/Decrypt DPL

Operational Test Data FAQ

Operational Test Data FAQ

Operational test data is the set of values a test consumes during execution.

Last updated: September 18, 2026