



# **DIMS Test Plan Documentation**

***Release 2.9.1***

**David Dittrich, Stuart Maclean, Linda Parsons**

October 02, 2016



<b>1</b>	<b>Scope</b>	<b>1</b>
1.1	Identification . . . . .	1
1.2	CSCI capability requirements . . . . .	1
1.3	System overview . . . . .	1
1.4	Software System Test Strategy (General) . . . . .	2
1.5	Document overview . . . . .	2
<b>2</b>	<b>Referenced documents</b>	<b>5</b>
<b>3</b>	<b>Software test environment</b>	<b>7</b>
3.1	UW Tower server room . . . . .	7
3.2	Software items . . . . .	7
3.3	Hardware and firmware items . . . . .	8
3.4	Proprietary nature, acquirer's rights, and licensing . . . . .	8
3.5	Installation, testing, and control . . . . .	8
3.6	Participating organizations . . . . .	9
3.7	Orientation plan . . . . .	9
3.8	Tests to be performed . . . . .	9
<b>4</b>	<b>Test identification</b>	<b>11</b>
4.1	General information . . . . .	11
4.2	General test conditions . . . . .	13
4.3	Planned tests . . . . .	13
<b>5</b>	<b>Test schedules</b>	<b>25</b>
<b>6</b>	<b>Requirements traceability</b>	<b>27</b>
<b>7</b>	<b>Notes</b>	<b>29</b>
7.1	Glossary of Terms . . . . .	29
7.2	List of Acronyms . . . . .	30
<b>8</b>	<b>License</b>	<b>33</b>
<b>9</b>	<b>Appendices</b>	<b>35</b>
9.1	Test Dashboard server VM . . . . .	35
9.2	Creating Tupelo server VM . . . . .	36
9.3	Installing Tupelo client . . . . .	36
9.4	Planning tests with JIRA . . . . .	36

9.5	Generating a Test Report with the report generation utility . . . . .	44
9.6	Using <code>bats</code> for Producing and Executing Tests . . . . .	48
9.7	Using DIMS Bash functions in Bats tests . . . . .	58
<b>10</b>	<b>Contact</b>	<b>61</b>

---

## Scope

---

### 1.1 Identification

This Software System Test Plan (release 2.9.1) documents the objectives, approach, and scope/limitations of the testing and evaluation of the Distributed Incident Management System (DIMS). It describes the logistics for executing the plan, including test environment needs, high level schedule, and resource mapping. It also identifies technical and schedule risks and mitigation plans. This plan will be utilized by the project team to properly scope, manage, and comprehend test efforts. This test plan is specific to software, as defined within the statement of work.

This Test Plan describes the Formal Qualification Test (FQT) activities and the environment in which those activities will take place for the DIMS deployed system. DIMS is composed of the following Computer Software Configuration Items (CSCIs):

### 1.2 CSCI capability requirements

DIMS is funded by the Department of Homeland Security under contract HSHQDC- 13-C-B0013.

The DIMS system is divided into the following high-level CSCI sets, per the acquisition contract.

CSCI	Label	Contract Item
Backend data stores	BDS	C.3.1.1
Dashboard web application	DWA	C.3.1.1
Data Integration and User Tools	DIUT	C.3.1.2
Vertical/Lateral Info. Sharing	VLIS	C.3.1.3

Reference [DIMS System Requirements v 2.9.0](#) establishes the DIMS software requirements. Reference [DIMS Operational Concept Description v 2.9.0](#) describes the operational concept (and open source design concept) for DIMS. Reference [DIMS Architecture Design v 2.9.0](#) sets forward the architectural design. Unit testing and integration testing will be performed during development at the University of Washington. This document primarily addresses the *FQT* activities that take place when deploying the DIMS software.

### 1.3 System overview

The primary mission objectives for the DIMS system are operational in nature, focused on facilitating the exchange of operational intelligence and applying this intelligence to more efficiently respond and recover from cyber compromise. The secondary mission objectives are to create a framework in which tools to support the primary mission objectives can more quickly and easily be integrated and brought to bear against advancing techniques on the attacker side of the equation.

The DIMS project is intended to take this semi-automated sharing of structured threat information, building on the success of the Public Regional Information Security Event Monitoring (PRISEM) project and leveraging an existing community of operational security professionals known as Ops- Trust, and scale it to the next level. The intent of this software project is to allow for near real-time sharing of critical alerts and structured threat information that will allow each contributing party to receive information, alerts and data, analyze the data, and respond appropriately and in a timely manner through one user-friendly web application, DIMS.

Working with the use cases defined by MITRE and PRISEM users, building the features necessary to simplify structured information sharing, and operationalizing these within these existing communities, will allow DIMS to fill existing gaps in capabilities and support existing missions that are slowed down today by many complicated, manual processes.

The changes to existing systems consists of seamless integration of the three current systems into a single web application that enables each system to contribute to the data warehouse of information concerning threats, alerts, attacks and suspect or compromised user terminals within the infrastructure. Additionally, the integrated systems will be able to share and retrieve data, visually observe alerts through color coded visual indicators, while retaining the existing functionality of the current system.

## 1.4 Software System Test Strategy (General)

The software system test strategy we will employ is compromises two high-level sets of tests.

The first set of test will be developed as the system is being designed using the Agile coding methodology. Development and pre-release testing will be performed during “sprints” (cycles of development anticipated to be on 2 week time frames). These tests are centered on development and take place within the project team. The project team will extract user stories from the use cases and in turn create epics derived from the user stories. Tests will be written for the user stories and determine the necessary development. This is essentially a test-driven approach that pushes development from one sprint to the next.

The other set of tests will be performed with interaction from the stakeholders and will entail an environment in which the stakeholders will, at determinate intervals, access the system and execute one or more user stories. This environment, set up specifically for stakeholders, will enable them to interact with the system at their discretion and direct their feedback to the project team about the functionality delivered within their environment.

Delivery of software for testing will be conducted in the following manner:

- Software for the project will be developed using the Agile Methodology and will therefore be delivered incrementally, based on completed functionality for each Sprint.
- Each Sprint cycle is anticipated to be two to four (2-4) weeks.
- A final feature complete build will be delivered at the completion of development.
- Testing efforts will be structured to test the delivered functionality of each Sprint’s delivered software module(s).
- Feature complete builds will be tested as integrated modules.
- Iterations of testing will be conducted as builds are delivered to address bug fixes as well as delivered features or functionality.

## 1.5 Document overview

The purpose of this document is to establish the requirements for the testing of the Distributed Incident Management System (DIMS). The structure of this document and the strategy for testing has been adapted principally from MIL-STD-498 (see Section [Referenced documents](#)). It contains the following information:

- Section [Referenced documents](#) lists related documents.

- Section *Software test environment* specifies the test environment that will be used in testing DIMS CSCIs. It includes a description of the hardware, software and personnel resources needed for installation, testing and control.
- Section *Test identification* provides general information about test levels and test classes, general test conditions, and planned tests.
- Section *Requirements traceability* describes traceability of tests back to requirements.
- Section *Notes* provides an alphabetical listing of acronyms and abbreviations used in this document.





---

## Referenced documents

---

1. DIMS System Requirements v 2.9.0
2. DIMS Operational Concept Description v 2.9.0
3. DIMS Architecture Design v 2.9.0
4. MIL-STD-498, Military Standard Software Development and Documentation, AMSC No. N7069, Dec. 1994.



---

## Software test environment

---

### 3.1 UW Tower server room

DIMS deployments are currently hosted in the University of Washington Tower (“UW Tower”) data center. Plans are in place to migrate hardware to another data center at [Semaphore Corporation](#) in Seattle.

Each deployed system will be separated from the others and operate independently (in terms of services and logical network topologies). Even if they are in the same rack and possibly sharing some fundamental resources (e.g., Network Attached Storage (NAS), or switched VLANs, they will each be deployed, configured, and will operate in such a manner as to be fully taken down without impacting the operation of the other environments.

This will allow development, test and evaluation, and user acceptance test

### 3.2 Software items

Table 3.1: Software Items Table

Software item	Purpose
<a href="#">Jira</a>	Ticketing, task tracking, etc.
<a href="#">Zephyr for Jira</a>	Test management plug-in for <a href="#">Jira</a>
<a href="#">Robot Framework</a>	Open source Acceptance Test-Driven Development (ATTD) tool
<a href="#">Bats</a>	<a href="#">TAP</a> -compliant testing framework for Bash
Custom DIMS scripts	Serve specific custom test functions as needed.

### 3.3 Hardware and firmware items

Table 3.2: Hardware Items Tables

Hardware Item	Purpose
<ul style="list-style-type: none"><li>• Dell PowerEdge R715 server</li><li>• 128GB RAM</li><li>• 2x12-Core 1.8GHz AMD Opteron processors</li><li>• 12 x 1TB drives in RAID 5 array</li></ul>	Server capable of running all DIMS components in containers
<ul style="list-style-type: none"><li>• Dell PowerEdge R720 server</li><li>• 128GB RAM</li><li>• 4x12-Core 2.40Ghz Intel Xeon E5-2695 v2</li><li>• 6 x 4TB drives in RAID 5 array</li></ul>	Server capable of running all DIMS components in containers
<ul style="list-style-type: none"><li>• Dell PowerEdge R510 server</li><li>• 2 x 1TB drives</li></ul>	Compute cluster capable server

### 3.4 Proprietary nature, acquirer's rights, and licensing

Some tests defined and anticipated under this plan involve use of a licensed [Jira](#) ticketing system using the [Zephyr for Jira](#) plug-in. These are primarily development-related tests that fall under the levels *Unit*, *Integration*, and/or *Component Interface* test levels, of type *Expected Value* and/or *Desk check* as defined in Section [Test levels](#) and [Test classes](#), respectively.

The remainder of the tests will use open source products either acquired from their original source, or produced by the DIMS team and delivered with the final system. (See Section [License](#) for the license under which DIMS software is being released.)

### 3.5 Installation, testing, and control

Deployment tests will start with a bare-metal server with a network connection capable of routing to the internet. From there, the following general high-level steps will be performed:

1. Operating system installation to the bare-metal server will be performed according to steps outlined in documentation. (This may be done using DHCP dynamically assigned addresses so as to minimize the number of manual steps required to install the base operating system.)
2. Network level configuration of the operating system will be performed by manually entering the required attributes (e.g., the way [Security Onion Setup Phase 1](#) is performed) into a software configuration database and/or configuration file.
3. The software configuration database and/or configuration file will be applied to the system, configuring all DIMS components for initial use.
4. Further manual steps will be necessary to provision initial user accounts in the portal and/or other DIMS system administration components.

5. The system will be put into a “test” mode to perform system tests to validate that all DIMS components are up and running and the system is functional. Initial data input tests may be performed at this point to validate that input and output functions are working.

## 3.6 Participating organizations

Table *Participants Roles* lists participants, their roles and responsibilities, related to testing.

Table 3.3: Participants Roles

Participants	Roles and Responsibilities
<b>DIMS development team</b>	Primary persons involved in testing DIMS components at all test levels. Most will be on-site at the specified test locations, while some will be remote.
<b>PRISEM participants</b>	Involved in higher-level testing focused on user acceptance testing and bug identification, as their time and availability permit. May be on-site or remote.
<b>Other stakeholders</b>	Involved in higher-level testing focused on user acceptance testing and bug identification, as their time and availability permit. Most likely will be remote.

## 3.7 Orientation plan

People involved with testing will be provided with guidance in the form of user documentation, a copy of this test plan, and any specifics of how to perform the specified tests. This may include providing them with access to software described in *Software items*, or some other form of checklist that will enable them to know the tests to be performed, acceptance criteria, and a means of reporting test results.

## 3.8 Tests to be performed

Specific tests of test classes *expected value testing* and *desk check testing* (see Section *Test classes*) that are manual in nature, and are expected to be performed by stakeholders for the purpose of acceptance testing, will be documented and provided as part of orientation prior to testing.



---

## Test identification

---

### 4.1 General information

Tests described in this section trace back to the [DIMS System Requirements v 2.9.0](#) document, Section [Requirements](#), as described in Section [Requirements traceability](#).

#### 4.1.1 Test levels

DIMS components will be tested at four distinct levels.

1. **Unit tests [U]**: These are tests of individual software components at the program or library level. These tests are primarily written by those who are developing the software to validate the software elements at a low (e.g., library or discrete shell command) perform their functions properly, independent of any other components.
2. **Integration tests [I]**: These are tests that are intended to verify the interfaces between components against the software design. Defects between interfaces are identified by these tests before their impact is observed at the system level through random or systemic failures.
3. **Component interface tests [C]**: These are checks of how data is processed as it is entered into and output from the system. Expected output may be compared against a cryptographic hash of the actual output to determine when actual output is malformed or otherwise deviates from expectations. Other interface tests (e.g., web application graphical user interface input/output) may be tested manually through visual inspection by a test user.
4. **System tests [S]**: Also known as *end-to-end* tests, these are tests to determine if the overall system meets its requirements for general data processing and function. All system components produce test results that are compiled into a single system test report that can be compared to detect differences between system tests, or to identify specific components that may have failed somewhere within the larger complex system.

The first two levels of tests are performed on a continuous basis during development. The final two levels pertain to the [FQT](#) described in this document.

---

**Note:** These **test levels** are to be identified in test related code and data using the following identifiers:

- unit [U]
  - integration [I]
  - component [C]
  - system [S]
-

### 4.1.2 Test classes

We will employ one or more of the following classes of tests to DIMS components:

1. **Expected value [EV]**: values from the expected classes of the input domain will be used to test nominal performance
2. **Simulated data [SD]**: simulated data for nominal and extreme geophysical conditions will be used to support error detection, recovery and reporting
3. **Erroneous input [EI]**: sample values known to be erroneous will be used to test error detection, recovery and reporting
4. **Stress [ST]**: maximum capacity of the input domain, including concurrent execution of multiple processes will be used to test external interfaces, error handling and size and execution time
5. **Timing [TT]**: wall clock time, CPU time and I/O time will be recorded
6. **Desk check [DC]**: both code and output will be manually inspected and analyzed

---

**Note:** These **test classes** are to be identified in test related code and data using the following identifiers:

- `expected_value [EV]`
  - `simulated_data [SD]`
  - `erroneous_input [EI]`
  - `stress [ST]`
  - `timing [TT]`
  - `desk_check [DC]`
- 

### 4.1.3 Qualification Methods

Five *qualification methods*<sup>1</sup> will be used in testing to establish conformance with requirements as described in this Section.

1. **Inspection**: Visual examination, review of descriptive documentation, and comparison of the actual characteristics with predetermined criteria.
2. **Demonstration**: Exercise of a sample of observable functional operations. This method is appropriate for demonstrating the successful integration, high-level functionality, and connectivity provided by the overall system.
3. **Manual Test**: Manual tests will be performed when automated tests are not feasible.
4. **Automated Test**: When possible, test procedures will be automated.
5. **Analysis**: Technical evaluation, processing, review, or study of accumulated data.

---

**Note:** These **qualification methods** are to be identified in test related code and data using the following identifiers:

- `inspection`
  - `demonstration`
  - `manual_test`
- 

<sup>1</sup> Source: [Automated Software Testing: Introduction, Management, and Performance](#), by Elfriede Dustin, Jeff Rashka, and John Paul.



- `automated_test`
  - `analysis`
- 

## 4.2 General test conditions

### 4.2.1 Data recording, reduction, and analysis

Test results from each test will be stored and indexed so as to be retrievable and post-processed for two primary reasons:

1. To be able to compare *TestA* to *TestB* and determine the difference in results (e.g., to identify regression errors, site-specific differences that were not anticipated during development, or uncover latent bugs related to services that are not managed properly and may not come up after a crash or other failure condition.
2. To be able to produce reStructuredText format files that can be inserted into a directory hierarchy for the Test Report document that can then be rendered using Sphinx to produce a deliverable HTML and/or PDF version.

This will allow developers to test code releases before they are pushed to “production” deployments, and for involved stakeholders doing independent field testing to generate test reports that can be sent back to the DIMS development team for debugging and code fixes.

## 4.3 Planned tests

### 4.3.1 Backend Data Stores CSCI - (BDS)

Backend data stores include temporary and long-term storage of event data, user attributes, user state, indicators and observables, and other incident response related data produced during use of the DIMS system. The following sections describe the scope of formal testing for the Backend Data Stores (BDS) CSCI.

#### Test Levels

General testing of the Backend Data Stores CSCI will take place at the levels described in *Test levels*. Unit and integration levels apply to development, and the remaining levels apply to *FQT*.

- Unit tests
- Integration tests
- Component interface tests
- System tests

#### Test Classes

The following classes of tests, described in *Test classes* will be performed during formal qualification testing of the Backend Data Stores CSCI:

- Expected value testing
- Simulated data
- Erroneous input

- Desk check testing

### General Test Conditions

The following sub-paragraphs identify and describe the planned collections of *FQT* tests. Test personnel should have access to the Firefox web browser, VPN access, a properly configured DIMS shell environment for testing.

### Acceptance Tests

This collection of tests are run by a Tester via the User Interface to exercise the Backend Data Stores CSCI and verify its functionality satisfies requirements in requirements and user stories. Acceptance tests will be entered, managed, executed, and reported via JIRA. The test descriptions, steps, test data, expected results for each step, and actual results will be included in the Test Report.

1. Test levels: System
2. Test type or class: Expected value, simulated data, erroneous input, desk check
3. Qualification method: Test
4. SR reference: [\[\[attributeStorage\]\] Attribute Storage](#), [\[\[bdsUserStory1\]\] BDS User Story 1](#), [\[\[bdsUserStory2\]\] BDS User Story 2](#),
5. Special requirements: Access to the DIMS JIRA tool
6. Type of data to be recorded: Tester, Execution date, Status (Pass/Fail)

### Operational Tests

Tests in the Operational collection are automated tests that run when the CSCI is started and at proscribed intervals during operation. These tests will report results via a log fanout and are used to verify system operation and availability. (Some of the test capabilities in this category will also be used for performance of the tests described in *States and Modes*.)

1. Test levels: System
2. Test type or class: Timing, desk check
3. Qualification method: Test
4. SR reference: [\[\[bdsUserStory1\]\] BDS User Story 1](#), [\[\[bdsUserStory2\]\] BDS User Story 2](#)
5. Type of data to be recorded: Component ID, Wall clock time, other data TBD.

## 4.3.2 Dashboard Web Application CSCI - (DWA)

The Dashboard Web Application, also referred to as the DIMS Dashboard, consists of web application server (“DWA Server”) and client (“DWA Client”) components. The following sections describe the scope of testing for the Dashboard Web Application CSCI.

### Test Levels

General testing of the Dashboard Web Application CSCI will take place at the levels described in *Test levels*. Unit and integration levels apply to development, and the remaining levels apply to *FQT*.

- Unit tests

- Integration tests
- Component interface tests
- System tests

## Test Classes

The following classes of tests, described in *Test classes* will be performed during formal qualification testing of the Dashboard Web Application CSCI:

- Expected value testing
- Simulated data
- Erroneous input
- Desk check testing

## General Test Conditions

The following sub-paragraphs identify and describe the planned collections of *FQT* tests. Test personnel should have access to the Firefox web browser, VPN access, a properly configured DIMS shell environment for testing.

### User Interface Tests

The purpose of this collection is to validate the functionality of Dashboard Web Application User Interface (UI) elements. UI tests will be entered, managed, executed, and reported via JIRA. The test descriptions, steps, test data, expected results for each step, and actual results will be included in the Test Report.

1. Test levels: Component interface
2. Test type or class: Expected value, simulated data, erroneous input, desk check
3. Qualification method: Test
4. SR reference: [\[\[dwaUserStory7\]\] DWA User Story 7](#)
5. Special requirements: Access to the DIMS JIRA tool
6. Type of data to be recorded: Tester, Execution date, Status (Pass/Fail)

### Acceptance Tests

This collection of tests are run by a Tester via the User Interface to exercise the Dashboard Web Application and verify its functionality satisfies requirements in user stories. Acceptance tests will be entered, managed, executed, and reported via JIRA. The test descriptions, steps, test data, expected results for each step, and actual results will be included in the Test Report.

1. Test levels: System
2. Test type or class: Expected value, simulated data, erroneous input, desk check
3. Qualification method: Test
4. SR reference: [\[\[dwaUserStory1\]\] DWA User Story 1](#), [\[\[dwaUserStory2\]\] DWA User Story 2](#), [\[\[dwaUserStory3\]\] DWA User Story 3](#), [\[\[dwaUserStory4\]\] DWA User Story 4](#), [\[\[dwaUserStory5\]\] DWA User Story 5](#), [\[\[dwaUserStory6\]\] DWA User Story 6](#), [\[\[dwaUserStory9\]\] DWA User Story 9](#)

5. Special requirements: Access to the DIMS JIRA tool
6. Type of data to be recorded: Tester, Execution date, Status (Pass/Fail)

### Operational Tests

Tests in the Operational collection are automated tests that run when the CSCI is started and at proscribed intervals during operation. These tests will report results via a log fanout and are used to verify system operation and availability. (Some of the test capabilities in this category will also be used for performance of the tests described in *States and Modes*.)

1. Test levels: System
2. Test type or class: Timing, desk check
3. Qualification method: Test
4. SR reference: [\[\[dwaUserStory8\]\] DWA User Story 8](#)
5. Type of data to be recorded: Component ID, Wall clock time, other data TBD.

### 4.3.3 Data Integration and User Tools CSCI - (DIUT)

The following sections describe the scope of formal testing for the Data Integration and User Tools (DIUT) CSCI.

#### Test Levels

General testing of the Data Integration and User Tools CSCI will take place at the levels described in *Test levels*. Unit and integration levels apply to development, and the remaining levels apply to *FQT*.

- Unit tests
- Integration tests
- Component interface tests
- System tests

#### Test Classes

The following classes of tests, described in *Test classes* will be performed during formal qualification testing of the Data Integration and User Tools CSCI:

- Expected value testing
- Simulated network failures testing
- Stress testing
- Timing testing

#### General Test Conditions

The following sub-paragraphs identify and describe the planned groups of tests for the DIUT CSCI.

### Tupelo Whole Disk Initial Acquisition Test

This test relates to Tupelo, a whole disk acquisition and search tool which is one component of the DIUT. The purpose of this test is to ensure that the entire contents of a test disk of arbitrary size can be uploaded to a Tupelo store component over a network.

1. Test Levels: integration, system
2. Test classes: expected value, timing, stress
3. Qualification Method: Demonstration, inspection
4. SR reference: [\[\[diutUserStory6\]\] DIUT User Story 6](#)
5. Type of Data Recorded: Copy of test disk content stored in Tupelo store.

### Tupelo Whole Disk Subsequent Acquisition Test

This test also relates to Tupelo. The purpose of this test is to ensure that the entire contents of a test disk of arbitrary size can be uploaded to a Tupelo store component over a network. That disk was previously uploaded to the same store. The upload time and filesystem usage at the store site should be less than for an initial upload.

1. Test Levels: integration, system
2. Test classes: expected value, timing
3. Qualification Method: Demonstration, inspection
4. SR reference: [\[\[diutUserStory6\]\] DIUT User Story 6](#)
5. Type of Data Recorded: Test log showing smaller stored disk and reduced elapsed time for disk acquisition.

### Tupelo Store Tools Test

This test also relates to Tupelo. The purpose of this test is to ensure that Tupelo store-processing tools can create so-called ‘products’ from previously uploaded disk images. These products are then to be stored in the same store as the images.

1. Test Levels: integration, system
2. Test classes: expected value, timing
3. Qualification Method: Demonstration, inspection
4. SR reference: [\[\[diutUserStory6\]\] DIUT User Story 6](#)
5. Type of Data Recorded: Products of store tools to exist as supplementary files in Tupelo store.

### Tupelo Artifact Search Test

This test also relates to Tupelo. The purpose of this test is to ensure that a search request sent to a Tupelo store, via e.g. AMQP, results in the correct response. If the search input identifies an artifact which should be found in the store, a positive result must be communicated to the search invoker. Similarly for a query which should be not located. The objective is to avoid false positives and false negatives.

1. Test Levels: integration, system
2. Test classes: expected value, timing
3. Qualification Method: Demonstration, inspection

4. SR reference: [\[\[diutUserStory6\]\] DIUT User Story 6](#)
5. Type of Data Recorded: Log files generated when making test queries of the existence of various files to a Tupelo store.

### Tupelo Sizing Test

This test also relates to Tupelo. The purpose of this test is to stress the Tupelo software by inputting a large disk image, on the order of 1 or even 2TB.

1. Test Levels: integration, system
2. Test classes: stress, timing
3. Qualification Method: Demonstration, inspection
4. SR reference: [\[\[diutUserStory6\]\] DIUT User Story 6](#)
5. Type of Data Recorded: Copy of test disk content stored in Tupelo store.

### Tupelo Network Failure Test

This test also relates to Tupelo. The purpose of this test is to assert the correctness of the Tupelo store when a disk upload is interrupted by both a client failure and a network failure.

1. Test Levels: integration, system
2. Test classes: expected state
3. Qualification Method: Demonstration, inspection
4. SR reference: [\[\[diutUserStory6\]\] DIUT User Story 6](#)
5. Type of Data Recorded: Summary of Tupelo store contents before and after a whole disk upload operation interrupted by a client or network failure.

### Tupelo Boot Media Test 1

This test also relates to Tupelo. The purpose of this test is to check that a computer can be booted from a CD/USB containing a Linux Live CD with integrated Tupelo software, and that the local hard drive(s) of that computer can be uploaded to a remote Tupelo store over the network.

1. Test Levels: integration, system
2. Test classes: expected state
3. Qualification Method: Demonstration, inspection
4. SR reference: [\[\[diutUserStory6\]\] DIUT User Story 6](#)
5. Type of Data Recorded: Observed behavior during demonstration.
6. Special Requirements: Tupelo Boot CD

### Tupelo Boot Media Test 2

This test also relates to Tupelo. The purpose of this test is to check that a computer can be booted from a CD/USB containing a Linux Live CD with integrated Tupelo software, and that the local hard drive(s) of that computer can be uploaded to a Tupelo store located on a locally attached external hard drive.

1. Test Levels: integration, system
2. Test classes: expected state
3. Qualification Method: Demonstration, inspection
4. SR reference: [\[\[diutUserStory6\]\] DIUT User Story 6](#)
5. Type of Data Recorded: Disk contents of computer's own hard drive and external hard drive.
6. Special Requirements: Tupelo Boot CD and External Hard Drive and Cabling

### User Interface Tests

The purpose of this collection is to validate the functionality of the Data Integration and User Tools capabilities related to general incident response and/or incident tracking or investigative activities. These tests are related to tests described in *User Interface Tests* in the DWA CSCI section. DIUT CSCI tests will be entered, managed, executed, and reported via JIRA. The test descriptions, steps, test data, expected results for each step, and actual results will be included in the Test Report.

1. Test levels: Component interface
2. Test type or class: Expected value, simulated data, erroneous input, desk check
3. Qualification method: Test
4. SR reference: [\[\[diutUserStory2\]\] DIUT User Story 2](#), [\[\[diutUserStory8\]\] DIUT User Story 8](#)
5. Special requirements: Access to the DIMS JIRA tool
6. Type of data to be recorded: Tester, Execution date, Status (Pass/Fail)

### Acceptance Tests

This collection of tests are run by a Tester via the User Interface to exercise the Data Integration and User Tools capabilities and verify its functionality satisfies requirements in user stories. These tests are related to tests described in *Acceptance Tests* in the DWA CSCI section. Acceptance tests will be entered, managed, executed, and reported via JIRA. The test descriptions, steps, test data, expected results for each step, and actual results will be included in the Test Report.

1. Test levels: System
2. Test type or class: Expected value, simulated data, erroneous input, desk check
3. Qualification method: Test
4. SR reference: [\[\[incidentTracking\]\] Incident/Campaign Tracking](#), [\[\[knowledgeAcquisition\]\] Knowledge Acquisition](#), [\[\[aggregateSummary\]\] Summarize Aggregate Data](#), [\[\[diutUserStory1\]\] DIUT User Story 1](#), [\[\[diutUserStory3\]\] DIUT User Story 3](#), [\[\[diutUserStory4\]\] DIUT User Story 4](#), [\[\[diutUserStory5\]\] DIUT User Story 5](#), [\[\[diutUserStory7\]\] DIUT User Story 7](#)
5. Special requirements: Access to the DIMS JIRA tool
6. Type of data to be recorded: Tester, Execution date, Status (Pass/Fail)

### Operational Tests

Tests in the Operational collection are automated tests that run when the CSCI is started and at proscribed intervals during operation. These tests will report results via a log fanout and are used to verify system operation and availability.

(Some of the test capabilities in this category will also be used for performance of the tests described in *States and Modes*.)

1. Test levels: System
2. Test type or class: Timing, desk check
3. Qualification method: Test
4. SR reference: [\[\[aggregateSummary\]\] Summarize Aggregate Data](#), [\[\[diutUserStory2\]\] DIUT User Story 2](#), [\[\[diutUserStory4\]\] DIUT User Story 4](#), [\[\[diutUserStory8\]\] DIUT User Story 8](#)
5. Type of data to be recorded: Component ID, Wall clock time, other data TBD.

### 4.3.4 Vertical/Lateral Information Sharing CSCI - (VLIS)

The following sections describe the scope of formal testing for the Vertical and Lateral Information Sharing (VLIS) CSCI.

#### Test Levels

General testing of the Vertical and Lateral Information Sharing CSCI will take place at the levels described in *Test levels*. Unit and integration levels apply to development, and the remaining levels apply to FQT.

- Unit tests
- Component interface tests
- System tests

#### Test Classes

The following classes of tests, described in *Test classes* will be performed during formal qualification testing of the Vertical and Lateral Information Sharing CSCI:

- Expected value testing

#### General Test Conditions

The following sub-paragraphs identify and describe the planned groups of tests.

#### Ingest of Indicators of Compromise via STIX Documents

This test relates to stix-java and Tupelo. stix-java is a DIMS-sourced Java library for manipulation of Mitre's STIX document format. STIX documents containing indicators-of-compromise (IOCs) in the form of file hashes and file names shall be parsed. The hashes and names shall be submitted to the DIMS Tupelo component, and all the stored disks searched for the IOCs. Hit or miss results are then collected.

1. Test Levels: component interface, system
2. Test classes: expected value
3. Qualification Method: Demonstration, inspection
4. SR reference: [\[\[structuredInput\]\] Structured data input](#)
5. Type of Data Recorded: Copy of search results, copy of input STIX documents, summary of Tupelo store state.



## Authoring of Indicators of Compromise via STIX Documents

This test relates to stix-java. stix-java is a DIMS-sourced Java library for manipulation of Mitre's STIX document format. STIX documents containing indicators-of-compromise (IOCs) in the form of file hashes and file names shall be created. The hashes and names shall be auto-generated from output of CIF feeds, from Ops-Trust email attachments and from Tupelo whole disk analysis results.

1. Test Levels: component interface, system
2. Test classes: expected value
3. Qualification Method: Demonstration, inspection
4. SR reference: `[[structuredInput]] Structured data input`
5. Type of Data Recorded: Copy of created STIX documents, summary of Tupelo store state, CIF feed results

### 4.3.5 States and Modes

There are several states/modes that the DIMS system must support, including a *test mode*, *debug mode*, and a *demonstration mode*. The following section describes the scope of testing for these states/modes.

#### Test Levels

General testing of the required states/modes will take place at the *System level* only, as described in *Test levels*.

#### Test Classes

The following classes of tests, described in *Test classes* will be performed during formal qualification testing of states/modes.

- Desk check testing

#### General Test Conditions

The following sub-paragraphs identify and describe the planned collections of *FQT* tests. Test personnel should have access to the Firefox web browser, VPN access, a properly configured DIMS shell environment for testing.

#### States/Modes Tests

The purpose of this collection is to validate the functionality of the defined states/modes. These tests will be entered, managed, executed, and reported via JIRA. The test descriptions, steps, test data, expected results for each step, and actual results will be included in the Test Report.

1. Test levels: System level
2. Test type or class: Desk check
3. Qualification method: Test
4. SR reference: `[[modeToggles]] Mode toggles`, `[[testMode]] Test Mode`, `[[debugMode]] Debug Mode`, `[[demoMode]] Demonstration Mode`
5. Special requirements: Access to the DIMS JIRA tool
6. Type of data to be recorded: Tester, Execution date, Status (Pass/Fail)

### 4.3.6 Security and Privacy Tests

There are several security controls related to user accounts, access keys, and network access. The following section describes the scope of testing for these aspects of DIMS.

#### Test Levels

General testing of the required security and privacy requirements will take place at the *Component interface level* and *System level*, as described in *Test levels*.

#### Test Classes

The following classes of tests, described in *Test classes* will be performed during formal qualification testing of states/modes.

- Expected value testing
- Erroneous input
- Desk check testing

#### General Test Conditions

The following sub-paragraphs identify and describe the planned collections of *FQT* tests. Test personnel should have access to the Firefox web browser, VPN access, a properly configured DIMS shell environment for some testing, while other tests (e.g., port scanning) will be done from external hosts without any proper account or credential data.

#### Security Tests

The purpose of this collection is to validate the functionality of the defined security and privacy requirements. These tests will be entered, managed, executed, and reported via JIRA. The test descriptions, steps, test data, expected results for each step, and actual results will be included in the Test Report.

1. Test levels: Component interface level, System level
2. Test type or class: Expected value, Erroneous Input, Desk check
3. Qualification method: Test
4. SR reference: `[[networkAccessControls]]` Network Access Controls, `[[accountAccessControls]]` Account Access Controls, `[[secondFactorAuth]]` Second-factor authentication, `[[accountSuspension]]` Account suspension, `[[keyRegeneration]]` Key Regeneration and Replacement
5. Special requirements: Access to the DIMS JIRA tool
6. Type of data to be recorded: Tester, Execution date, Status (Pass/Fail)

#### Operational Tests

Tests in the Operational collection are automated tests that run on-demand or at proscribed intervals during normal operation. These tests will report results via both the DWA CSCI components, and a log fanout and are used to verify system operation and availability. (Some of the test capabilities in this category are closely related to tests described in *Operational Tests*.)

1. Test levels: System

2. Test type or class: Timing, desk check
3. Qualification method: Test
4. SR reference: [\[\[diutUserStory2\]\] DIUT User Story 2](#), [\[\[diutUserStory4\]\] DIUT User Story 4](#), [\[\[diutUserStory5\]\] DIUT User Story 5](#)
5. Type of data to be recorded: Component ID, Wall clock time, other data TBD.

---

**Note:** An application penetration test of DIMS components, including the *Dashboard Web Application CSCI - (DWA)* and the ops-trust portal (part of *Vertical/Lateral Information Sharing CSCI - (VLIS)* and described in DIMS Operational Concept Description v 2.9.0, Sections Ops-Trust portal Code Base and Ops-Trust portal Code Base) is to be performed by a professional service company.

This is a separate test from those described in this Test Plan, and the results will be reported in a separate document to be included in the final Test Report.

---

### 4.3.7 Design and Implementation Tests

A set of contractual requirements deal with the design and implementation of the internal software system and documentation. Tests in this collection are manual tests based on inspection or other observational qualification methods.

1. Test levels: System
2. Test type or class: Desk check
3. Qualification method: Manual Test, Inspection
4. SR reference: [\[\[automatedProvisioning\]\] Automated Provisioning](#), [\[\[agileDevelopment\]\] Agile development](#), [\[\[continuousIntegration\]\] Continuous Integration & Delivery](#), [\[\[leverageOpenSource\]\] Leveraging open source components](#)
5. Type of data to be recorded: Declarative statements as appropriate.

### 4.3.8 Software Release Tests

A set of contractual requirements deal with the public release of open source software components and documentation. Tests in this collection are manual tests based on inspection or other observational qualification methods.

1. Test levels: System
2. Test type or class: Desk check
3. Qualification method: Manual Test, Inspection
4. SR reference: [\[\[exportControl\]\] Export control](#), [\[\[noEncryption\]\] No included cryptographic elements](#), [\[\[open-SourceRelease\]\] Open source release](#)
5. Type of data to be recorded: Declarative statements as appropriate.



---

## Test schedules

---

To date, two sets of tests at all levels and classes as defined in Section *Test identification*. Each of these tests took a significant effort from all team members in order to perform and report on the results.

Due to the amount of effort required, full tests to the same extent as previous tests will only be performed when absolutely necessary and within available staff resources.

Because of the amount of effort required, more automated tests in the **component**, **interface**, and **system** levels (primarily in the form of **expected\_value** class tests) that can be integrated into the system provisioning and Ansible configuration processes to assist with regression testing and acceptance testing of bugfix and feature branches prior to making new releases. The use of the TAP-compliant *Bats* program, and *Robot Framework* (see Section *Software Items Table*), will eventually allow an automated test and report generation flow that could support more frequent testing with less staff time required to perform the testing.



---

## Requirements traceability

---

Traceability to requirements in this document is done in Section *Test identification* using `intersphinx` document linking as part of the *SR reference* line in test descriptions. This creates hypertext links to reference the requirements and/or user stories from the appropriate sub-section(s) of Section *Requirements* from the document *DIMS System Requirements v 2.9.0*, like this:

...

1. Test levels: Component interface
2. Test type or class: Expected value, simulated data, erroneous input, desk check
3. Qualification method: Test
4. SR reference: [\[\[dwaUserStory7\]\] DWA User Story 7](#)
5. Special requirements: Access to the DIMS JIRA tool
6. Type of data to be recorded: Tester, Execution date, Status (Pass/Fail)

...

---

**Note:** The links only work when viewing the PDF and/or HTML versions of this document, with an active network connection.

---





---

## Notes

---

This document is structured on MIL-STD-498, described at [A forgotten military standard that saves weeks of work](#) (by providing free project management templates), by Kristof Kovacs. Specifically, this document is modeled on [STP.html](#).

## 7.1 Glossary of Terms

**Agile** A programming methodology based on short cycles of feature-specific changes and rapid delivery, as opposed to the “Waterfall” model of system development with long requirements definition, specification, design, build, test, acceptance, delivery sequences of steps.

**Botnets System** The name given to the re-implementation of *Einstein 1* technology. See <http://web.archive.org/web/20131115180654/http://www.botnets.org/>

**cron** A Unix/Linux service daemon that is responsible for running background tasks on a scheduled basis.

**Git** A source code version management system in widespread use.

**CIFglue** “Simple rails app to quickly add indicators to the Collective Intelligence Framework”

**Cryptographic Hash, Cryptographic Hashing Algorithm** A mathematical method of uniquely representing a stream of bits with a fixed-length numeric value in a numeric space sufficiently large so as to be infeasible to predictably generate the same hash value for two different files. (Used as an integrity checking mechanism). Commonly used algorithms are MD5, SHA1, SHA224, SHA256, RIPEMD-128. (See also [http://en.wikipedia.org/wiki/Cryptographic\\_hash\\_function](http://en.wikipedia.org/wiki/Cryptographic_hash_function)).

**Einstein 1** A network flow based behavioral and watchlist based detection system developed by University of Michigan and Merit Networks, Inc. for use by US-CERT. The re-implementation is known as the *Botnets System*.

**Fusion Center** Entities created by DHS to integrate federal law enforcement and intelligence resources with state and local law enforcement for greater collaboration and information sharing across levels of SLTT governments.

**GZIP** Gnu ZIP (file compression program)

**MUTEX** Mutual Exclusion (object or lock, used to synchronize execution of independent threads or processes that must share a common resource in an exclusive manner, or to ensure only one copy of a program is running at a time)

**NetFlow** Record format developed by Cisco for logging and storing Network Flow information (see also SiLKTools).

**NoSQL** The term for database that does not use the typical table-based relational schema as Relational Database Management Systems (RDBMS)

**Ops-Trust (ops-t)** Operational Security Trust organization (see <http://ops-trust.net/>)

**Redis** A “NoSQL” database system used to store files in a key/value pair model via a RESTful HTTP/HTTPS interface.

**SiLKTools** A network flow logging and archiving format and tool set developed by Carnegie Mellon’s Software Engineering Institute (in support of CERT/CC).

**Team Cymru** (Pronounced “COME-ree”) – “Team Cymru Research NFP is a specialized Internet security research firm and 501(c)3 non-profit dedicated to making the Internet more secure. Team Cymru helps organizations identify and eradicate problems in their networks, providing insight that improves lives.”

**Tupelo** A host-based forensic system (client and server) developed at the University of Washington, based on the HoneyNet Project “Manuka” system.

## 7.2 List of Acronyms

**AAA** Authentication, Authorization, and Accounting

**AMQP** Advanced Message Queuing Protocol

**AS** Autonomous System

**ASN** Autonomous System Number

**CI** Critical Infrastructure

**CIDR** Classless Internet Domain Routing

**CIF** Collective Intelligence Framework

**CIP** Critical Infrastructure Protection

**CISO** Chief Information and Security Officer

**COA** Course of Action (steps to Respond and Recover)

**CONOPS** Concept of Operations

**CRADA** Cooperative Research and Development Agreement

**CSC** Computer Software Component

**CSCI** Computer Software Configuration Item

**CSIRT** Computer Security Incident Response Team

**CSV** Comma-separated Value (a semi-structured file format)

**DIMS** Distributed Incident Management System

**DNS** Domain Name System

**DoS** Denial of Service

**DDoS** Distributed Denial of Service

**EO** Executive Order

**FQT** Formal Qualification Test/Tests/Testing

**HSPD** Homeland Security Presidential Directive

**ICT** Information and Communication Technology

**IOC** Indicators of Compromise

**IP** Internet Protocol (TCP and UDP are examples of Internet Protocols)

**IRC** Internet Relay Chat (an instant messaging system)

**JSON** JavaScript Object Notation

**MAPP** Microsoft Active Protections Program

**MNS** Mission Needs Statement

**NCFTA** National Cyber-Forensics & Training Alliance

**NTP** Network Time Protocol (a service exploited to perform reflected/amplified DDoS attacks by spoofing the source address of requests, where the much larger responses flood the victim)

**OODA** Observe, Orient, Decide, and Act (also known as the “Boyd Cycle”)

**PPD** Presidential Policy Directive

**PRISEM** Public Regional Information Security Event Management

**RBAC** Role Based Access Control

**RESTful** Representational State Transfer web service API

**RPC** Remote Procedure Call

**SCADA** Supervisory Control and Data Acquisition

**SIEM** Security Information Event Management (sometimes referred to as Security Event Information Management, Security Event Monitoring, causing some to pronounce it as “sim-sem”).

**SLTT** State, Local, Territorial, and Tribal (classification of non-federal government entities)

**SOC** Security Operations Center

**SoD** Security on Demand (PRISEM project support vendor)

**SSH** Secure Shell

**STIX** Structure Threat Information Expression. A standard for information exchange developed by MITRE in support of DHS US-CERT.

**TAXII** Trusted Automated Exchange of Indicator Information

**TCP** Transmission Control Protocol (one of the Internet Protocols)

**TLP** Traffic Light Protocol

**TTP** Tools, Tactics, and Procedures

**UC** Use Case

**UDP** Unreliable Datagram Protocol (one of the Internet Protocols)

**WCX** Western Cyber Exchange



---

## License

---

*Section author: Dave Dittrich (@davedittrich) <dittrich@u.washington.edu>*

```
Berkeley Three Clause License
=====
```

```
Copyright (c) 2014, 2015 University of Washington. All rights reserved.
```

```
Redistribution and use in source and binary forms, with or without
modification, are permitted provided that the following conditions are met:
```

1. Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.
2. Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.
3. Neither the name of the copyright holder nor the names of its contributors may be used to endorse or promote products derived from this software without specific prior written permission.

```
THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS" AND
ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED
WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE
DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT HOLDER OR CONTRIBUTORS BE LIABLE
FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL
DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR
SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER
CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY,
OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE
OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
```



---

## Appendices

---

This appendix describes:

1. How to create a VM for Dashboard server and API component interface tests, Dashboard UI-based tests
2. How to create a Tupelo server VM for Tupelo tests
3. How to install Tupelo client
4. How to plan tests with JIRA
5. How to use the test report generation scripts in the `$GIT/dims-tr` repo.
6. How to use `bats` for producing and executing tests

### 9.1 Test Dashboard server VM

Tests performed using the Dashboard user interface, as well as some of the automated tests, require a Vagrant test server VM running on the tester's workstation.

Whenever a test has a prerequisite of `dashboard test server`, it is referring to this VM, up and running on the tester's workstation.

#### 9.1.1 Creating the VM

To create this VM, do the following:

1. The Vagrant box `ubuntu-14.04.2-amd64-base-keyed` should be installed in Vagrant. This box is created by the procedure described at `dimspacker:vmquickstart`. If you have already performed those steps, through the subsection `dimspacker:vmquickstartinstallboxes`, then you've already installed this box file. You can also check by:

```
vagrant box list
```

If `ubuntu-14.04.2-amd64-base-keyed` appears in the list, then you already have this box file installed, and you can go to the next step. If it is not installed, perform the steps at `dimspacker:vmquickstart` through the box installation step at `dimspacker:vmquickstartinstallboxes`.

2. Set up Vagrant directory for the VM you're creating. You will need to name the VM - the name `dimstestserver` is used in the code below and will be used in Test prerequisite steps. Then you will add a private IP address to the auto-generated Vagrantfile. We are specifying the IP to be used for testing as `'192.168.56.103'`. Finally, bring the VM up using `vagrant up`.

```
cd $GIT/dims-vagrant/ubuntu-14.04.2-amd64/  
make server NAME=dimstestserver  
cd dimstestserver  
../nic2 192.168.56.103  
vagrant up
```

3. Run the Ansible playbook `dims-test-server-provision.yml` as shown below. This will take a while to run since it will install the virtual environment. Once this is done, you can reuse the VM for multiple tests without destroying it.

```
./run_playbook -g dashboard-test-servers dims-test-server-provision.yml -vv
```

## 9.1.2 Resetting VM Data

A test prerequisite may specify that you reset the test VM's data. To do this, you run the `dashboard-test-data-reset` playbook as follows. CD to the VM's Vagrantfile directory as shown in the first step if you aren't there already.

```
cd $GIT/dims-vagrant/ubuntu-14.04.2-amd64/dimstestserver  
./run_playbook -g dashboard-test-servers dashboard-test-data-reset.yml -vv
```

## 9.2 Creating Tupelo server VM

Tupelo tests require a Tupelo server VM.

This server is created in a similar fashion to the dashboard test server:

1. Make sure you have the box file installed as shown above in step one of creating a dashboard test server.
2. Set up Vagrant directory for the VM, name it `tupelosever` and give it an IP of `192.168.56.102`.

```
cd $GIT/dims-vagrant/ubuntu-14.04.2-amd64/  
make server NAME=tupelosever  
cd tupelosever  
../nic2 192.168.56.102  
vagrant up
```

3. Run the Ansible playbook `tupelo-test-server-provision.yml` as shown below.

```
./run_playbook -g tupelo-servers tupelo-server-install.yml -vv
```

## 9.3 Installing Tupelo client

The tester needs the Tupelo client installed on their host machine to perform many of the Tupelo tests. The tester installs the tupelo client on his/her developer workstation via Ansible:

```
RUNHOST=localhost RUNGROUP=tupelo-clients ansible-playbook -i $GIT/ansible-playbooks/dyn_inv.py $GIT/
```

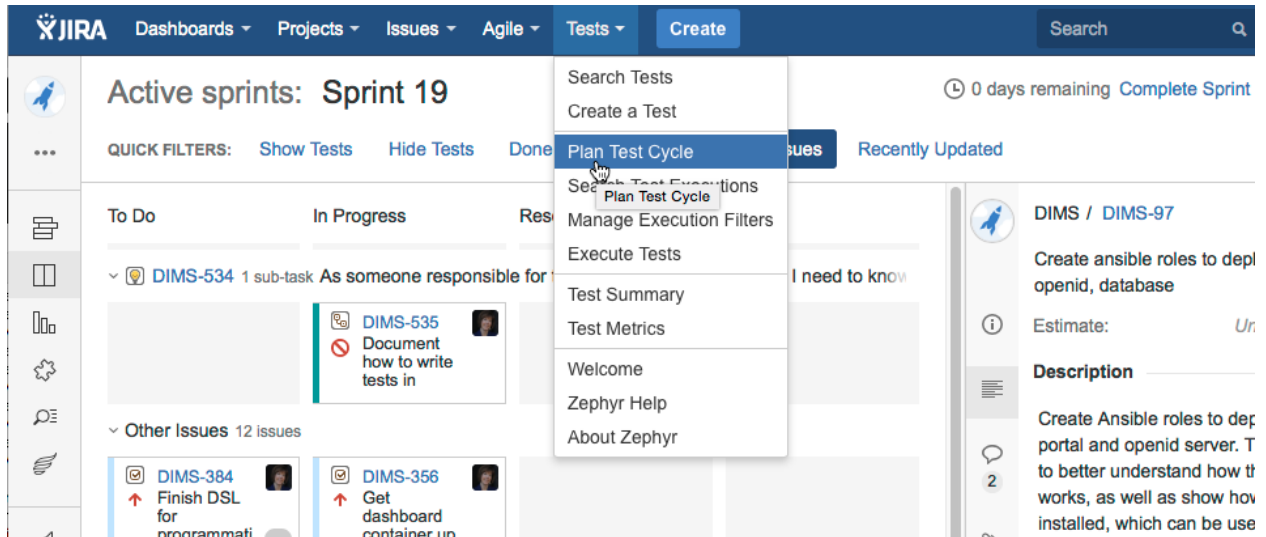
## 9.4 Planning tests with JIRA

This section describes how to plan a test cycle and write tests using JIRA.

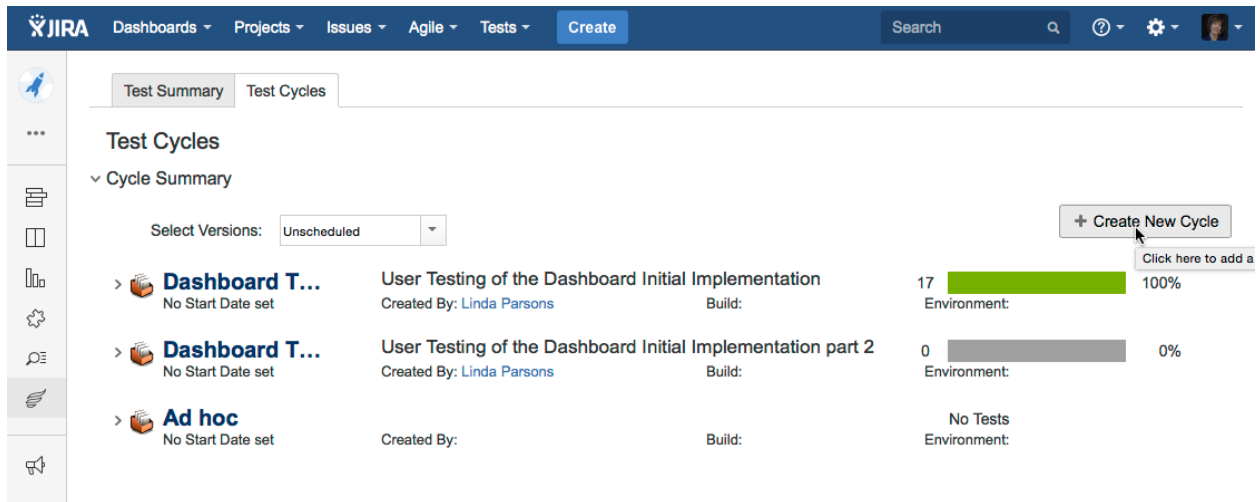


### 9.4.1 Test cycle

We use a test cycle to plan and execute our tests. To view test cycles, click `Tests > Plan Test Cycle`:



The list of cycles displays. We need a new cycle for the tests due on 11/15, so we'll create one. Click the `Create New Cycle` button to bring up a dialog to create the cycle. Give it a name and description.



The new test cycle displays in the list. You can see that it doesn't have any tests yet.

When you create a test in JIRA, you will add it to this test cycle.

### 9.4.2 Creating tests

To create a new test, select `Tests > Create a Test`.

The Create Issue screen displays, with the issue type already set to Test. Enter a summary for the test, and fill in choices in the testLevel, testClass, and qualificationMethod pick boxes. These are described in this Test Plan. Choose one item per pick box. You should also add the reference to the DIMS SR - these are referenced in Section 4 of this

**Test Cycles**

Test Summary | Test Cycles

▼ Cycle Summary

Select Versions: Unscheduled + Create New Cycle

Test Cycle	Description	Tests	Environment
> <b>Dashboard T...</b>	User Testing of the Dashboard Initial Implementation Created By: <a href="#">Linda Parsons</a> Build:	17	100%
> <b>Dashboard T...</b>	User Testing of the Dashboard Initial Implementation part 2 Created By: <a href="#">Linda Parsons</a> Build:	0	0%
> <b>2015-11-15_t...</b>	Tests planned for the test report due on 11/15/2015 Created By: <a href="#">Linda Parsons</a> Build:	No Tests	Environment:
> <b>Ad hoc</b>	No Start Date set Created By: Build:	No Tests	Environment:

**JIRA** Dashboards ▾ Projects ▾ Issues ▾ Agile ▾ Tests ▾ Create Search

**TESTS** <<

**PREDEFINED FILTERS**

- My Executed Tests
- My Failed Executions
- All Unexecuted Tests
- All Executed Tests
- All Failed Executions

**MANAGE EXECUTION FILTERS**

- Manage Execution Filters

**SEARCH EXECUTION**

Search Save Save

project = "DIMS" AND fixVersion = "2.9.1"

in ("2015-11-15\_test\_report")

g the request.

**Tests Menu:**


- Search Tests
- Create a Test**
- Plan Test Cycle
- Search Test Executions
- Manage Execution Filters
- Execute Tests
- Test Summary
- Test Metrics
- Welcome
- Zephyr Help
- About Zephyr

plan for each group of planned tests. typeOfData describes where the output data will be when you are done. You can add this later if you don't know it at this time.

The following figure shows the first part of the Create Issue dialog being filled in:


### Create Issue

Project **DIMS**

Issue Type  **Test**

Summary\*

Priority ↑ Should Have (Major) ?

Due Date  

Epic Link

Choose an epic to assign this issue to.

testLevel 

None  
component\_interface  
integration  
system  
unit

Select test levels for this test

testClass 

None  
expected\_value  
simulated\_data  
erroneous\_input  
stress

Test Classes as per DIMS Test Plan

qualificationMethod 

None  
inspection  
demonstration  
manual\_test  
automated\_test

Qualification methods as per DIMS Test Plan

dimssrReference

Test traces back to this reference

typeOfData

Data produced by the test - where it is, what kind of data

Scrolling down, you describe the Environment and provide a Description of the test. The environment entry should be short. If the test needs a local Vagrant VM to run, then the Test should reference how that is created in the prerequisites.

We enter prerequisites in the first test step. When you initially create the test, you can just add a short description and add prerequisites by editing the test.

Save the test. You can further fill out fields by editing the test. For example, you can upload files needed to run the test. In this example, we are uploading a file with test data and a script which will run a number of automated tests

Fix Version/s **None**

Assignee  [Assign to me](#)

Reporter\*

Start typing to get a list of possible matches.

Environment

Style
B
I
U
A
A
Link
Image
List
List
Emoji
+

Local test system (Ubuntu 14) running via Vagrant, created via Test DIMS-XXX (TBA)



For example operating system, software platform and/or hardware specifications (include as appropriate for the issue).

Description

Style
B
I
U
A
A
Link
Image
List
List
Emoji
+

Tests getting and setting attributes for users via the dashboard server REST interface.



Original Estimate  (eg. 3w 4d 12h) ?

The original estimate of how much work is involved in resolving this issue.

Remaining Estimate  (eg. 3w 4d 12h) ?

An estimate of how much work remains until this issue will be resolved.

Attachment

Drop files here to attach them

or

Select files

Labels

Begin typing to find and create labels or press down to select a suggested label.

[Create](#) [Cancel](#)

using the test data file as input:

Edit Issue : DIMS-545

Configure Fields

Other specific requirements - like having VPN connected

What scripts are needed and their location (git or attached to the ticket for example)

What input data files are needed and their location (git or attached to the ticket for example)

For example:

Original Estimate

(eg. 3w 4d 12h)

The original estimate of how much work is involved in resolving this issue.

Remaining Estimate

(eg. 3w 4d 12h)

An estimate of how much work remains until this issue will be resolved.

Attachment

Drop files here to attach them

or

Select files

test1\_script.sh

input\_data1

Labels

Begin typing to find and create labels or press down to select a suggested label.

Comment

Style B I U A A Link Image List Bulleted List Emoji +

Update

Cancel

If files aren't attached, the prerequisites should state where to get them.

Add the test to the desired test cycle. Select **More Actions** > **Add to Test Cycle(s)**:

Select the test cycle. In this example, we choose the Sample Test Cycle. You would choose the 2015-100-15\_test\_report test cycle for actual tests.

The test will now show up in the list of tests for that test cycle. The E button on the right is the button to click when you are going to execute the test.

To create more tests, you can do so from scratch, or you can clone an existing test. Go to the existing test, and click Clone.

Enter a new summary for the new test. You can clone attachments if the same ones are used for the new test.

**JIRA** Dashboards ▾ Projects ▾ Issues ▾ Agile ▾ Tests ▾ **Create**

DIMS / DIMS-545

## Sample test 1 for 11/15 test plan

**Edit** **Comment** **Admin** ▾ **Clone** **More Actions** ▾ **Execute**

**Details**

Type: Test

Priority: Should Have

Component/s: None

Labels: None

Environment: Tester dev laptop with local test VM provisioned as per XXX

Test Levels: component\_interface

Test Classes: expected\_value

Qualification Methods: automated\_test

Test Areas: BDS

Status: **OPEN** (View Workflow)

Resolution: Unresolved

**Description**

Sample test to show an automated test in JIRA

Prerequisites: List the prerequisites here to run the test.

### Add to Test Cycle(s)

Version: **Unscheduled** ▾

Test Cycle: 

- ✓ Ad hoc
- Dashboard Test Cycle 1
- Dashboard Test Cycle 2
- 2015-11-15\_test\_report
- Sample Test Cycle**

☐ Add to more Test Cycle(s) **Add** **Cancel**


Search   Export Tools

project = "DIMS" AND fixVersion = "Unscheduled" AND cycleName in ("Sample Test Cycle")

1-1 of 1



<input type="checkbox"/> Cycle Name	Issue Key	Test Summary	Project Name	Priority	Execution Defect(s)	Executed By	Executed On	Creation Date
<input type="checkbox"/> Sample Test Cycle	DIMS-545	Sample test 1 for 11/15 test plan	DIMS	Should Have (Major)				9/Nov/15

1-1 of 1


 DIMS / DIMS-545

## Sample test 1 for 11/15 test plan

**Details**

Type:	 Test	Status:	<b>OPEN</b> (View Wc
Priority:	 Should Have (Major)	Resolution:	Unresolved
Component/s:	None		
Labels:	None		
Environment:	Tester dev laptop with local test VM provisioned as per XXX		
Test Levels:	component_interface		
Test Classes:	expected_value		
Qualification Methods:	automated_test		
Test Areas:	BDS		

### Clone

 Enter the summary of the clone issue ...


Summary\*

☒ Clone Attachments

Press Ctrl+Alt+s to submit this

Here is an updated Sample test 2. Prerequisite info has been added to the description. The comment regarding “if test fails” isn’t needed - that was put in before we had the typeOfOutput field (will update this screenshot later);

Since this test is automated, we just have one step - to run the test script. The Expected Result is given as how many tests should pass.


DIMS / DIMS-547

## Sample test 2 for 11/15 test plan

Edit
Comment
Admin
Clone
More Actions
Execute

### Details

Type:	Test	Status:	<b>OPEN</b> <a href="#">(View Workflow)</a>
Priority:	Should Have (Major)	Resolution:	Unresolved
Component/s:	None		
Labels:	None		
Environment:	Tester dev laptop with local test VM provisioned as per XXX		
testLevel:	component_interface		
testClass:	expected_value		
qualificationMethod:	automated_test		
dimssrReference:	dimssr:bdsstory1		
typeOfData:	file: /path/to/file		
Test Areas:	BDS		

### Description

Sample test to show an automated test in JIRA.

If test fails, attach test report or add its location to comments for the test run.

Prerequisites:

1. Developer VM connected to the VPN, running current dims-ci-utils and current python virtual environment
2. Local test VM created as per DIMS-566 (url). VM should be up and running
3. Test script attached to this ticket: test1\_script.sh
4. Test data attached to this ticket: input\_data1

Click to edit

### Test Details

	Test Step	Test Data	Expected Result
1	Run script test1_script.sh:	input_data2, attached to this ticket. It is provided	Passed 12 Failed 0

### People

Assign

Repo

Votes

Watch

### Dates

Create

Update

### Agile

View

### HipChat

Discuss

Connect

## 9.5 Generating a Test Report with the report generation utility

### Note:

A Test Report is produced using a reporting utility that generates a Sphinx document, based on source from the \$GIT/dims-tr Git repository directory. It processes metadata descriptions of tests and their results, producing a Sphinx document.



Test Details

	Test Step	Test Data	Expected Result		Comment
1	Run script test1_script.sh:	input_data1, attached to this ticket. It is provided to the test script	Passed 14 Failed 0 Total 14		Test step was executed and failed.
	bash ./test1_script.sh input_data1	"test1_script.sh" as an argument.			Enter Comment

PASS

FAIL

WIP

BLOCKED

✓ UNEXECUTED

✓

✗

## Execute Test: Testcase

All steps are updated to **FAIL**. Do you also want to update current test status to

FAIL

↑

↓

Execute

Cancel

Basically - everything the tester needs to know to run the test.

Test Execution

Execution Status: **FAIL**

Executed By: Linda Parsons

Executed On: Today 6:23 PM

Defects:

Comment:

Attachments (Execution)

Test Details

	Test Step	Test Data	Expected Result	Status	Comment
1	Run script test1_script.sh:	input_data1, attached to this ticket. It is provided to the test script	Passed 14 Failed 0 Total 14	<b>FAIL</b>	Enter Comment
	bash ./test1_script.sh input_data1	"test1_script.sh" as an argument.			

Sample Test Cycle

No Start Date set

Sample Test Cycle  
Created By: Linda Parsons

Build:

Environment:

1  50%

ID	Status	Summary	Defect	Component	Label	Executed By	Executed On
DIMS-547	UNEXECUTED	Sample test 2 for 11/15 test plan					
DIMS-545	<b>FAIL</b>	Sample test 1 for 11/15 test plan				Linda Parsons	Today 6:53 PM

<b>Sample Test Cycle</b> No Start Date set		<b>Sample Test Cycle</b> Created By: Linda Parsons		Build:	Environment:	1	<div><div></div></div> 50%
ID	Status	Summary	Defect	Component	Label	Executed By	Executed On
DIMS-547	UNEEXECUTED	Sample test 2 for 11/15 test plan					
DIMS-545	FAIL	Sample test 1 for 11/15 test plan				Linda Parsons	Today 6:53 PM

Test cycles are named with a date, e.g., 2015-11-15. A simple directory structure is used that combines test results from both test managed within Jira using Zephyr for Jira, as well as non-Jira test results. Both of these sources are rooted at `$GIT/dims-tr/test_cycles` along with a copy of the Sphinx document skeleton found in the `$GIT/dims-tr/docs` directory. This separates the report and data from which the report was generated for each test cycle into its own directory tree. For example,

```
[dimscli] dittrich@27b:~/dims/git/dims-tr/test_cycles
(feature/dims-529*) $ tree
.
+-- 2015-11-15
|   +-- docs
|   |   +-- Makefile
|   |   +-- build
|   |   +-- source
|   +-- jira_data
|   |   +-- DIMS-553.json
|   |   +-- DIMS-553.pdf
|   |   +-- DIMS-554.json
|   |   +-- DIMS-554.pdf
|   |   +-- DIMS-565.json
|   |   +-- DIMS-565.pdf
|   |   +-- DIMS-566.json
|   |   +-- DIMS-569.json
|   |   +-- DIMS-570.json
|   |   +-- DIMS-570.pdf
|   |   +-- DIMS-571.json
|   |   +-- DIMS-571.pdf
|   |   +-- DIMS-574.json
|   |   +-- DIMS-574.pdf
|   +-- jira_data_summary.json
|   +-- nonjira_data
|   |   +-- test1.json
|   |   +-- test1.pdf
|   |   +-- test2.json
|   |   +-- test3.json
+-- 2016_00_00
|   +-- docs
|   |   +-- Makefile
|   |   +-- build
|   |   +-- source
|   +-- jira_data
|   |   +-- blahblah
|   +-- jira_data_summary.json
|   +-- nonjira_data
|   |   +-- blahblah
20 directories, 91 files
```

**Note:** The directory 2016\_00\_00 is just an example to show two sub-trees, not just one: it does not exist in Git.

A file `$GIT/dims-tr/CURRENT_CYCLE` contains the test cycle identifier for the current test cycle (and can be over-ridden with a command line option in the test utility.)

It can be used with inline command substitution in the BASH shell like this:

```
[dimscli] dittrich@27b:~/dims/git/dims-tr (feature/dims-529*) $ tree -L
1 test_cycles/$(cat CURRENT_CYCLE)
test_cycles/2015-11-15
+-- docs
+-- jira_data
+-- jira_data_summary.json
+-- nonjira_data

3 directories, 1 file

[dimscli] dittrich@27b:~/dims/git/dims-tr (feature/dims-529*) $ tree -L
1 test_cycles/$(cat CURRENT_CYCLE)/jira_data
test_cycles/2015-11-15/jira_data
+-- DIMS-553.json
+-- DIMS-553.pdf
+-- DIMS-554.json
+-- DIMS-554.pdf
+-- DIMS-565.json
+-- DIMS-565.pdf
+-- DIMS-566.json
+-- DIMS-569.json
+-- DIMS-570.json
+-- DIMS-570.pdf
+-- DIMS-571.json
+-- DIMS-571.pdf
+-- DIMS-574.json
+-- DIMS-574.pdf

0 directories, 14 files
```

There is a helper Makefile at the root of the repo to make it easier to generate a report.

```
dimscli] dittrich@27b:~/dims/git/dims-tr (feature/dims-529*) $ make help
/Users/dittrich/dims/git/dims-tr
[Using Makefile.dims.global v1.6.124 rev ]
-----
Usage: make [something]

Where "something" is one of the targets listed in the sections below.

-----
Targets from Makefile.dims.global

help - Show this help information (usually the default rule)
dimsdefaults - show default variables included from Makefile.dims.global
version - show the Git revision for this repo
envcheck - perform checks of requirements for DIMS development
-----
Targets from Makefile

all - defaults to 'report'
showcurrent - show the current test cycle
enter - enter a test description
```

```
report - generate a 'results.rst' file in ../docs/source/
autobuild - run dims.sphinxautobuild for this test cycle
install - install Python script and pre-requisites
clean - remove build files and generated .rst files.
spotless - clean, then also get rid of dist/ directory
-----

[dimscli] dittrich@27b:~/dims/git/dims-tr (feature/dims-529*) $ make
showcurrent
Current test cycle is 2015-11-15

[dimscli] dittrich@27b:~/dims/git/dims-tr (feature/dims-529*) $ make report
python scripts/get_test.py

[dimsenv] dittrich@27b:~/dims/git/dims-tr (feature/dims-529*) $ make
autobuild
tar -cf - docs | (cd "test_cycles/2015-11-15" && tar -xf -)
rm -rf build/*
[I 151119 21:35:18 server:271] Serving on http://127.0.0.1:48196
[I 151119 21:35:18 handlers:58] Start watching changes
[I 151119 21:35:18 handlers:60] Start detecting changes

+----- source/test3.rst changed
-----
/Users/dittrich/dims/git/dims-tr/test_cycles/2015-11-15/docs/source/test3.rst::
WARNING: document isn't included in any toctree
+-----

+----- source/index.rst changed
-----
+-----

+----- source/results.rst changed
-----
+-----

[I 151119 21:35:24 handlers:131] Browser Connected: http://127.0.0.1:48196/
```

## 9.6 Using bats for Producing and Executing Tests

The DIMS project has adopted use of the [Bats: Bash Automated Testing System](#) (known as `bats`) to perform simple tests in a manner that produces parsable output following the [Test Anything Protocol](#) (TAP).

Bats is a **TAP Producer**, whose output can be processed by one of many [TAP Consumers](#), including the Python program `tap.py`.

### 9.6.1 Organizing Bats Tests

This section covers the basic functionality of `bats` and how it can be used to produce test results.

We should start by looking at the `--help` output for `bats` to understand how it works in general.

```
$ bats -h
Bats 0.4.0
Usage: bats [-c] [-p | -t] <test> [<test> ...]

<test> is the path to a Bats test file, or the path to a directory
containing Bats test files.

-c, --count      Count the number of test cases without running any tests
-h, --help      Display this help message
-p, --pretty     Show results in pretty format (default for terminals)
-t, --tap        Show results in TAP format
-v, --version    Display the version number

For more information, see https://github.com/sstephenson/bats
```

As is seen, multiple tests – files that end in `.bats` – can be passed as a series of arguments on the command line. This can be either individual arguments, or a wildcard shell expression like `*.bats`.

If the argument evaluates to being a directory, `bats` will look through that directory and run all files in it that end in `.bats`.

**Caution:** As we will see, `bats` has some limitations that do not allow mixing file arguments and directory arguments. You can either give `bats` one or more files, or you can give it one or more directories, but you **cannot** mix files and directories.

To see how this works, let us start with a simple example that has tests that do nothing other than report success with their name. In this case, test `a.bats` looks like this:

```
#!/usr/bin/env bats

@test "a" {
    [[ true ]]
}
```

We produce three such tests, each in their own directory, following this organizational structure:

```
$ tree tests
tests
+-- a
|   +-- a.bats
+-- b
|   +-- b.bats
|   +-- c
|       +-- c.bats
3 directories, 3 files
```

Since the hierarchy shown here does not contain tests itself, but rather holds directories that in turn hold tests, how does we run the tests?

Running `bats` with an argument that includes the highest level of the directory hierarchy does not work to run any of the tests in subordinate directories:

```
$ bats tests
0 tests, 0 failures
```

Running `bats` and passing a directory that contains files with names that end in `.bats` runs all of the tests in *that* directory.

```
$ bats tests/a
a

1 test, 0 failures
```

If we specify the next directory `tests/b`, then `bats` will run the tests in that directory that end in `.bats`, but will not traverse down into the `tests/b/c/` directory.

```
$ bats tests/b
b

1 test, 0 failures
```

To run the tests in the lowest directory, that specific directory must be given on the command line:

```
$ bats tests/b/c
c

1 test, 0 failures
```

Attempting to pass all of the directories along as arguments does not work, as seen here:

```
$ bats tests/a /tests/b tests/b/c
bats: /tmp/b does not exist
/usr/local/Cellar/bats/0.4.0/libexec/bats-exec-suite: line 20: let: count+=: syntax error: operand ex
```

This means that we *can* separate tests into subdirectories, to any depth or directory organizational structure, as needed, but tests must be run on a per-directory basis, or identified and run as a group of tests passed as file arguments using wildcards:

```
$ bats tests/a/*.bats tests/b/*.bats tests/b/c/*.bats
a
b
c

3 tests, 0 failures
```

Because specifying wildcards in this way, with arbitrary depths in the hierarchy of directories below `tests/` is too hard to predict, use a program like `find` to identify tests by name (possibly using wildcards or `grep` filters for names), passing the results on to a program like `xargs` to invoke `bats` on each identified test:

```
$ find tests -name '*.bats' | xargs bats
1..3
ok 1 a
ok 2 b
ok 3 c
```

---

**Note:** Note that the output changed from the examples above, which include the arrow (“”) character, to now include the word `ok` instead in TAP format. This is because the default for terminals (i.e., a program that is using a TTY device, not a simple file handle to something like a pipe). To get the pretty-print output, add the `-p` flag, like this:

```
$ find tests -name '*.bats' | xargs bats -p
a
b
c

3 tests, 0 failures
```

---

A more realistic test is seen here. This file, `pycharm.bats`, is the product of a Jinja template that is installed by Ansible along with the [PyCharm Community Edition Python IDE](#).

```
#!/usr/bin/env bats
#
# Ansible managed: /home/dittrich/dims/git/ansible-playbooks/v2/roles/pycharm/templates/./templates
#
# vim: set ts=4 sw=4 tw=0 et :

load helpers

@test "[S][EV] Pycharm is not an installed apt package." {
    ! is_installed_package pycharm
}

@test "[S][EV] Pycharm Community edition is installed in /opt" {
    results=$(ls -d /opt/pycharm-community-* | wc -l)
    echo $results >&2
    [ $results -ne 0 ]
}

@test "[S][EV] \"pycharm\" is /opt/dims/bin/pycharm" {
    assert "pycharm is /opt/dims/bin/pycharm" type pycharm
}

@test "[S][EV] /opt/dims/bin/pycharm is a symbolic link to installed pycharm" {
    [ -L /opt/dims/bin/pycharm ]
}

@test "[S][EV] Pycharm Community installed version number is 2016.2.3" {
    assert "2016.2.3" bash -c "file $(which pycharm) | sed 's|\\(.*pycharm-community-\\)\\([^\]*\\)\\(\\./"
}
```

```
$ test.runner --level system --match pycharm
[+] Running test system/pycharm
[S][EV] Pycharm is not an installed apt package.
[S][EV] Pycharm Community edition is installed in /opt
[S][EV] "pycharm" is /opt/dims/bin/pycharm
[S][EV] /opt/dims/bin/pycharm is a symbolic link to installed pycharm
[S][EV] Pycharm Community installed version number is 2016.2.3

5 tests, 0 failures
```

## 9.6.2 Organizing tests in DIMS Ansible Playbooks Roles

The DIMS project uses a more elaborate version of the above example, which uses a *drop-in* model that allows any Ansible role to drop its own tests into a structured hierarchy that supports fine-grained test execution control. This drop-in model is implemented by the `tasks/bats-tests.yml` task playbook.

To illustrate how this works, we start with an empty test directory:

```
$ tree /opt/dims/tests.d
/opt/dims/tests.d

0 directories, 0 files
```

The base role has the largest number of tests, since it does the most complex foundational setup work for DIMS computer systems. The `template/tests` directory is filled with Jinja template Bash scripts and/or `bats` tests, in

a hierarchy that includes subdirectories for each of the defined test levels from Section [Test levels](#).

```
$ tree base/templates/tests
base/templates/tests
+-- component
+-- helpers.bash.j2
+-- integration
+-- README.txt
+-- system
|   +-- deprecated.bats.j2
|   +-- dims-accounts.bats.j2
|   +-- dims-accounts-sudo.bats.j2
|   +-- dims-base.bats.j2
|   +-- dns.bats.j2
|   +-- proxy.bats.j2
|   +-- sudo
|       +-- sudo-iptables.bats.j2
|   +-- user
|       +-- vpn.bats.j2
+-- unit
    +-- dims-filters.bats.j2

6 directories, 11 files
```

After running just the base role, the highlighted subdirectories that correspond to each of the test levels are now present in the `/opt/dims/tests.d/` directory:

```
$ tree /opt/dims/tests.d/
/opt/dims/tests.d/
+-- component
|   +-- helpers.bash -> /opt/dims/tests.d/helpers.bash
+-- helpers.bash
+-- integration
|   +-- helpers.bash -> /opt/dims/tests.d/helpers.bash
+-- system
|   +-- deprecated.bats
|   +-- dims-accounts.bats
|   +-- dims-accounts-sudo.bats
|   +-- dims-base.bats
|   +-- dims-ci-utils.bats
|   +-- dns.bats
|   +-- helpers.bash -> /opt/dims/tests.d/helpers.bash
|   +-- iptables-sudo.bats
|   +-- proxy.bats
|   +-- user
|       +-- helpers.bash -> /opt/dims/tests.d/helpers.bash
|       +-- vpn.bats
+-- unit
    +-- bats-helpers.bats
    +-- dims-filters.bats
    +-- dims-functions.bats
    +-- helpers.bash -> /opt/dims/tests.d/helpers.bash

5 directories, 18 files
```

Here is the directory structure for tests in the docker role:

```
/docker/templates/tests
+-- system
```



```

+-- docker-consul.bats.j2
+-- docker-core.bats.j2
+-- docker-network.bats.j2

1 directories, 3 files

```

If we now run the docker role, it will drop these files into the system subdirectory:

```

$ dims.ansible-playbook --role docker

PLAY [Ansible (2.x / v2) Base Playbook] *****

TASK [docker : include] *****
included: /home/dittrich/dims/git/ansible-playbooks/v2/tasks/pre_tasks.yml for dimsdemo1.devops.deve

. . .

PLAY RECAP *****
dimsdemo1.devops.develop : ok=34   changed=20   unreachable=0   failed=0

```

There are now 3 additional files (see emphasized lines for the new additions):

```

$ tree /opt/dims/tests.d
/opt/dims/tests.d
+-- component
|   +-- helpers.bash -> /opt/dims/tests.d/helpers.bash
+-- helpers.bash
+-- integration
|   +-- helpers.bash -> /opt/dims/tests.d/helpers.bash
+-- system
|   +-- deprecated.bats
|   +-- dims-accounts.bats
|   +-- dims-accounts-sudo.bats
|   +-- dims-base.bats
|   +-- dims-ci-utils.bats
|   +-- dns.bats
|   +-- docker-consul.bats
|   +-- docker-core.bats
|   +-- docker-network.bats
|   +-- helpers.bash -> /opt/dims/tests.d/helpers.bash
|   +-- iptables-sudo.bats
|   +-- proxy.bats
|   +-- user
|       +-- helpers.bash -> /opt/dims/tests.d/helpers.bash
|       +-- vpn.bats
+-- unit
    +-- bats-helpers.bats
    +-- dims-filters.bats
    +-- dims-functions.bats
    +-- helpers.bash -> /opt/dims/tests.d/helpers.bash

5 directories, 21 files

```

Tests can now be run by level, multiple levels at the same time, or more fine-grained filtering can be performed using find and grep filtering.

### 9.6.3 Running Bats Tests Using the DIMS `test.runner`

A test runner script (creatively named `test.runner`) is available to This script builds on and extends the capabilities of scripts like `test_runner.sh` from the GitHub [docker/swarm/test/integration](#) repository.

```
$ base/templates/tests/test.runner --help
usage: test.runner [options] args
flags:
  -d,--[no]debug:  enable debug mode (default: false)
  -E,--exclude:    tests to exclude (default: '')
  -L,--level:      test level (default: 'system')
  -M,--match:      regex to match tests (default: '.*')
  -l,--[no]list-tests: list available tests (default: false)
  -t,--[no]tap:    output tap format (default: false)
  -S,--[no]sudo-tests: perform sudo tests (default: false)
  -T,--[no]terse:  print only failed tests (default: false)
  -D,--testdir:    test directory (default: '/opt/dims/tests.d/')
  -u,--[no]usage:  print usage information (default: false)
  -v,--[no]verbose: be verbose (default: false)
  -h,--help:      show this help (default: false)
```

To see a list of all tests under a given test level, specify the level using the `--level` option. (The default is `system`). The following example shows a list of all the available `system` level tests:

```
$ test.runner --list-tests
system/dims-base.bats
system/pycharm.bats
system/dns.bats
system/docker.bats
system/dims-accounts.bats
system/dims-ci-utils.bats
system/deprecated.bats
system/coreos-prereqs.bats
system/user/vpn.bats
system/proxy.bats
```

To see all tests under any level, use `*` or a space-separated list of levels:

```
$ test.runner --level "*" --list-tests
system/dims-base.bats
system/pycharm.bats
system/dns.bats
system/docker.bats
system/dims-accounts.bats
system/dims-ci-utils.bats
system/deprecated.bats
system/coreos-prereqs.bats
system/user/vpn.bats
system/proxy.bats
unit/dims-filters.bats
unit/bats-helpers.bats
```

Certain tests that require elevated privileges (i.e., use of `sudo`) are handled separately. To list or run these tests, use the `--sudo-tests` option:

```
$ test.runner --list-tests --sudo-tests
system/dims-accounts-sudo.bats
system/iptables-sudo.bats
```

A subset of the tests can be selected using the `--match` option. To see all tests that include the word `dims`, do:

```
$ test.runner --level system --match dims --list-tests
system/dims-base.bats
system/dims-accounts.bats
system/dims-ci-utils.bats
```

The `--match` option takes an `egrep` expression to filter the selected tests, so multiple substrings (or regular expressions) can be passed with pipe separation:

```
$ test.runner --level system --match "dims|coreos" --list-tests
system/dims-base.bats
system/dims-accounts.bats
system/dims-ci-utils.bats
system/coreos-prereqs.bats
```

There is a similar option `--exclude` that filters out tests by `egrep` regular expression. Two of the four selected tests are then excluded like this:

```
$ test.runner --level system --match "dims|coreos" --exclude "base|utils" --list-tests
system/dims-accounts.bats
system/coreos-prereqs.bats
```

## 9.6.4 Controlling the Amount and Type of Output

The default for the `bats` program is to use `--pretty` formatting when standard output is being sent to a terminal. This allows the use of colors and characters like `and` to be used for passed and failed tests (respectively).

```
$ bats --help

[No write since last change]
Bats 0.4.0
Usage: bats [-c] [-p | -t] <test> [<test> ...]

<test> is the path to a Bats test file, or the path to a directory
containing Bats test files.

-c, --count      Count the number of test cases without running any tests
-h, --help      Display this help message
-p, --pretty     Show results in pretty format (default for terminals)
-t, --tap       Show results in TAP format
-v, --version    Display the version number

For more information, see https://github.com/sstephenson/bats

Press ENTER or type command to continue
```

We will limit the tests in this example to just those for `pycharm` and `coreos` in their names. These are relatively small tests, so it is easier to see the effects of the options we will be examining.

```
$ test.runner --match "pycharm|coreos" --list-tests
system/pycharm.bats
system/coreos-prereqs.bats
```

The `DIMS test.runner` script follows this same default output style of `bats`, so just running the two tests above gives the following output:

```
$ test.runner --match "pycharm|coreos"
[+] Running test system/pycharm.bats
[S][EV] Pycharm is not an installed apt package.
[S][EV] Pycharm Community edition is installed in /opt
[S][EV] "pycharm" is /opt/dims/bin/pycharm
[S][EV] /opt/dims/bin/pycharm is a symbolic link to installed pycharm
[S][EV] Pycharm Community installed version number is 2016.2.2

5 tests, 0 failures
[+] Running test system/coreos-prereqs.bats
[S][EV] consul service is running
[S][EV] consul is /opt/dims/bin/consul
[S][EV] 10.142.29.116 is member of consul cluster
[S][EV] 10.142.29.117 is member of consul cluster
[S][EV] 10.142.29.120 is member of consul cluster
[S][EV] docker overlay network "ingress" exists
[S][EV] docker overlay network "app.develop" exists
  (from function `assert' in file system/helpers.bash, line 18,
   in test file system/coreos-prereqs.bats, line 41)
  `assert 'app.develop' bash -c "docker network ls --filter driver=overlay | awk '/app.develop/ {
expected: "app.develop"
actual:   ""
[S][EV] docker overlay network "data.develop" exists
  (from function `assert' in file system/helpers.bash, line 18,
   in test file system/coreos-prereqs.bats, line 45)
  `assert 'data.develop' bash -c "docker network ls --filter driver=overlay | awk '/data.develop/
expected: "data.develop"
actual:   ""

8 tests, 2 failures
```

To get TAP compliant output, add the `--tap` (or `-t`) option:

```
$ test.runner --match "pycharm|coreos" --tap
[+] Running test system/pycharm.bats
1..5
ok 1 [S][EV] Pycharm is not an installed apt package.
ok 2 [S][EV] Pycharm Community edition is installed in /opt
ok 3 [S][EV] "pycharm" is /opt/dims/bin/pycharm
ok 4 [S][EV] /opt/dims/bin/pycharm is a symbolic link to installed pycharm
ok 5 [S][EV] Pycharm Community installed version number is 2016.2.2
[+] Running test system/coreos-prereqs.bats
1..8
ok 1 [S][EV] consul service is running
ok 2 [S][EV] consul is /opt/dims/bin/consul
ok 3 [S][EV] 10.142.29.116 is member of consul cluster
ok 4 [S][EV] 10.142.29.117 is member of consul cluster
ok 5 [S][EV] 10.142.29.120 is member of consul cluster
ok 6 [S][EV] docker overlay network "ingress" exists
not ok 7 [S][EV] docker overlay network "app.develop" exists
# (from function `assert' in file system/helpers.bash, line 18,
# in test file system/coreos-prereqs.bats, line 41)
# `assert 'app.develop' bash -c "docker network ls --filter driver=overlay | awk '/app.develop/ { p
# expected: "app.develop"
# actual:   ""
not ok 8 [S][EV] docker overlay network "data.develop" exists
# (from function `assert' in file system/helpers.bash, line 18,
# in test file system/coreos-prereqs.bats, line 45)
```

```
# `assert 'data.develop' bash -c "docker network ls --filter driver=overlay | awk '/data.develop/ {
# expected: "data.develop"
# actual:  ""
```

When running a large suite of tests, the total number of individual tests can get very large (along with the resulting output). To increase the signal to noise ratio, you can use the `--terse` option to filter out all of the successful tests, just focusing on the remaining failed tests. This is handy for things like validation of code changes and regression testing of newly provisioned Vagrant virtual machines.

```
$ test.runner --match "pycharm|coreos" --terse
[+] Running test system/pycharm.bats

5 tests, 0 failures
[+] Running test system/coreos-prereqs.bats
[S][EV] docker overlay network "app.develop" exists
  (from function `assert' in file system/helpers.bash, line 18,
  in test file system/coreos-prereqs.bats, line 41)
  `assert 'app.develop' bash -c "docker network ls --filter driver=overlay | awk '/app.develop/ {
  expected: "app.develop"
  actual:  ""
[S][EV] docker overlay network "data.develop" exists
  (from function `assert' in file system/helpers.bash, line 18,
  in test file system/coreos-prereqs.bats, line 45)
  `assert 'data.develop' bash -c "docker network ls --filter driver=overlay | awk '/data.develop/ {
  expected: "data.develop"
  actual:  ""

8 tests, 2 failures
```

Here is the same examples as above, but this time using the TAP compliant output:

```
$ test.runner --match "pycharm|coreos" --tap
[+] Running test system/pycharm.bats
1..5
ok 1 [S][EV] Pycharm is not an installed apt package.
ok 2 [S][EV] Pycharm Community edition is installed in /opt
ok 3 [S][EV] "pycharm" is /opt/dims/bin/pycharm
ok 4 [S][EV] /opt/dims/bin/pycharm is a symbolic link to installed pycharm
ok 5 [S][EV] Pycharm Community installed version number is 2016.2.2
[+] Running test system/coreos-prereqs.bats
1..8
ok 1 [S][EV] consul service is running
ok 2 [S][EV] consul is /opt/dims/bin/consul
ok 3 [S][EV] 10.142.29.116 is member of consul cluster
ok 4 [S][EV] 10.142.29.117 is member of consul cluster
ok 5 [S][EV] 10.142.29.120 is member of consul cluster
ok 6 [S][EV] docker overlay network "ingress" exists
not ok 7 [S][EV] docker overlay network "app.develop" exists
# (from function `assert' in file system/helpers.bash, line 18,
# in test file system/coreos-prereqs.bats, line 41)
# `assert 'app.develop' bash -c "docker network ls --filter driver=overlay | awk '/app.develop/ {
# expected: "app.develop"
# actual:  ""
not ok 8 [S][EV] docker overlay network "data.develop" exists
# (from function `assert' in file system/helpers.bash, line 18,
# in test file system/coreos-prereqs.bats, line 45)
# `assert 'data.develop' bash -c "docker network ls --filter driver=overlay | awk '/data.develop/ {
# expected: "data.develop"
```

```
# actual: ""

$ test.runner --match "pycharm|coreos" --tap --terse
[+] Running test system/pycharm.bats
1..5
[+] Running test system/coreos-prereqs.bats
1..8
not ok 7 [S][EV] docker overlay network "app.develop" exists
# (from function `assert' in file system/helpers.bash, line 18,
#   in test file system/coreos-prereqs.bats, line 41)
#   `assert 'app.develop' bash -c "docker network ls --filter driver=overlay | awk '/ap
# expected: "app.develop"
# actual:  ""

not ok 8 [S][EV] docker overlay network "data.develop" exists
# (from function `assert' in file system/helpers.bash, line 18,
#   in test file system/coreos-prereqs.bats, line 45)
#   `assert 'data.develop' bash -c "docker network ls --filter driver=overlay | awk '/d
# expected: "data.develop"
# actual:  ""
```

Figure `vagrantTestRunner` shows the output of `test.runner --level system --terse` at the completion of provisioning of two Vagrants. The one on the left has passed all tests, while the Vagrant on the right has failed two tests. Note that the error result has been passed on to `make`, which reports the failure and passes it along to the shell (as seen by the red `$` prompt on the right, indicating a non-zero return value).

```

[+] New VirtualBox VMs:
[+] Output saved to make-provision-201609181609.txt
[+] Running 'test.runner --level system --exclude vpn --terse' on vagrant:
[+] Running test system/dns-accounts.bats
1 test, 0 failures

[+] Running test system/dns.bats
17 tests, 0 failures

[+] Running test system/proxy.bats
1 test, 0 failures

[+] Running test system/deprecated.bats
1 test, 0 failures

[+] Running test system/docker.bats
7 tests, 0 failures

[+] Running test system/dins-base.bats
36 tests, 0 failures

X [S][EV] docker overlay network "app.local" exists
  (from function 'assert' in file system/helpers.bash, line 18)
    in test file system/coreos-prereqs.bats, (line 57)
      assert 'app.local' bash -c 'docker network ls --filter driver=overlay | awk '/app.local/ { print $2; }'" failed
    expected: "app.local"
    actual: ""

X [S][EV] docker overlay network "data.local" exists
  (from function 'assert' in file system/helpers.bash, line 18)
    in test file system/coreos-prereqs.bats, (line 41)
      assert 'data.local' bash -c 'docker network ls --filter driver=overlay | awk '/data.local/ { print $2; }'" failed
    expected: "data.local"
    actual: ""

7 tests, 2 failures

[+] Running test system/dins-ci-utils.bats
2 tests, 0 failures

Connection to 127.0.0.1 closed.
make[1]: *** [provision] Error 1
make[1]: Leaving directory '/vm/run/yellow'
make[1]: [reprovision-local] Error 2
(dmsenv) dittrich@dmsdemo1:/vm/run/yellow ( ) S
*** @OSGREGORY_HAYES@***** 20160918-3: 4- 5- 6- 7- 8-

```

## 9.7 Using DIMS Bash functions in Bats tests

The DIMS project Bash shells take advantage of a library of functions that are installed by the `base` role into `$DIMS/bin/dims_functions.sh`.

Bats has a pre- and post-test hooking feature that is very tersely documented (see [setup and teardown: Pre- and post-test hooks](#)):

You can define special setup and teardown functions, which run before and after each test case, respectively. Use these to load fixtures, set up your environment, and clean up when you're done.

What this means is that if you define a `setup()` function, it will be run *before* every `@test`, and if you define a `teardown()` function, it will be run *after* every `@test`.

We can take advantage of this to source the common DIMS `dims_functions.sh` library, making any defined functions in that file available to be called directly in a `@TEST` the same way it would be called in a Bash script.

An example of how this works can be seen in the unit tests for the `dims_functions.sh` library itself.

```

1  #!/usr/bin/env bats
2  #
3  # Ansible managed: /home/dittrich/dims/git/ansible-playbooks/v2/roles/base/templates/.../templates/t
4  #
5  # vim: set ts=4 sw=4 tw=0 et :
6
7  load helpers
8
9  function setup() {
10     source $DIMS/bin/dims_functions.sh
11 }
12
13 @test "[U][EV] say() strips whitespace properly" {
14     assert '[+] unce, tice, fee times a madie...' say '    unce,    tice,        fee times a madie..'
15 }
16
17 # This test needs to directly source dims_functions in bash command string because of multi-command
18 @test "[U][EV] add_on_exit() saves and get_on_exit() returns content properly" {
19     assert "'([0]=\"cat /dev/null\")'" bash -c ". $DIMS/bin/dims_functions.sh; touch /tmp/foo; add_
20 }
21
22 @test "[U][EV] get_hostname() returns hostname" {
23     assert "$(hostname)" get_hostname
24 }
25
26 @test "[U][EV] is_fqdn host.category.deployment returns success" {
27     is_fqdn host.category.deployment
28 }
29
30 @test "[U][EV] is_fqdn host.subdomain.category.deployment returns success" {
31     is_fqdn host.subdomain.category.deployment
32 }
33
34 @test "[U][EV] is_fqdn 12345 returns failure" {
35     ! is_fqdn 12345
36 }
37
38 @test "[U][EV] parse_fqdn host.category.deployment returns 'host category deployment'" {
39     assert "host category deployment" parse_fqdn host.category.deployment
40 }
41
42 @test "[U][EV] get_deployment_from_fqdn host.category.deployment returns 'deployment'" {
43     assert "deployment" get_deployment_from_fqdn host.category.deployment
44 }
45
46 @test "[U][EV] get_category_from_fqdn host.category.deployment returns 'category'" {

```

```

47     assert "category" get_category_from_fqdn host.category.deployment
48 }
49
50 @test "[U][EV] get_hostname_from_fqdn host.category.deployment returns 'host'" {
51     assert "host" get_hostname_from_fqdn host.category.deployment
52 }
53
54 @test "[U][EV] plural_s returns 's' for 0" {
55     assert "s" plural_s 0
56 }
57
58 @test "[U][EV] plural_s returns '' for 1" {
59     assert "" plural_s 1
60 }
61
62 @test "[U][EV] plural_s returns 's' for 2" {
63     assert "s" plural_s 2
64 }

```

**Attention:** Note that there is one test, shown on lines 17 through 20, that has multiple commands separated by semicolons. That compound command sequence needs to be run as a single command string using `bash -c`, which means it is going to be run as a new sub-process to the `assert` command line. Sourcing the functions in the outer shell does not make them available in the sub-process, so that command string must itself also source the `dims_functions.sh` library in order to have the functions defined at that level.



---

### Contact

---

*Section author: Dave Dittrich <dittrich @ u.washington.edu>, Stuart Maclean <stuart @ apl.washington.edu>, Linda Parsons <linda.parsons @ nextcentury.com>*



## A

AAA, [30](#)  
Agile, [29](#)  
AMQP, [30](#)  
AS, [30](#)  
ASN, [30](#)

## B

Botnets System, [29](#)

## C

CI, [30](#)  
CIDR, [30](#)  
CIF, [30](#)  
CIFglue, [29](#)  
CIP, [30](#)  
CISO, [30](#)  
COA, [30](#)  
CONOPS, [30](#)  
CRADA, [30](#)  
cron, [29](#)  
Cryptographic Hash, [29](#)  
Cryptographic Hashing Algorithm, [29](#)  
CSC, [30](#)  
CSCI, [30](#)  
CSIRT, [30](#)  
CSV, [30](#)

## D

DDoS, [30](#)  
DIMS, [30](#)  
DNS, [30](#)  
DoS, [30](#)

## E

Einstein 1, [29](#)  
EO, [30](#)

## F

FQT, [30](#)

Fusion Center, [29](#)

## G

Git, [29](#)  
GZIP, [29](#)

## H

HSPD, [30](#)

## I

ICT, [30](#)  
IOC, [30](#)  
IP, [30](#)  
IRC, [31](#)

## J

JSON, [31](#)

## M

MAPP, [31](#)  
MNS, [31](#)  
MUTEX, [29](#)

## N

NCFTA, [31](#)  
NetFlow, [29](#)  
NoSQL, [29](#)  
NTP, [31](#)

## O

OODA, [31](#)  
Ops-Trust (ops-t), [29](#)

## P

PPD, [31](#)  
PRISEM, [31](#)

## R

RBAC, [31](#)  
Redis, [30](#)

RESTful, [31](#)  
RPC, [31](#)

## **S**

SCADA, [31](#)  
SIEM, [31](#)  
SiLKTools, [30](#)  
SLTT, [31](#)  
SOC, [31](#)  
SoD, [31](#)  
SSH, [31](#)  
STIX, [31](#)

## **T**

TAXII, [31](#)  
TCP, [31](#)  
Team Cymru, [30](#)  
TLP, [31](#)  
TTP, [31](#)  
Tupelo, [30](#)

## **U**

UC, [31](#)  
UDP, [31](#)

## **W**

WCX, [31](#)