Table of Contents

Author: Martin Bergljung

- Introduction
- Architecture
- Concepts
  - Events
  - Event Processors
  - Event Results
  - Event Producers
  - Data Mirrors
  - Driver Event Processing
- Test Library (out-of-the-box tests)
  - Sample
  - Sign Up
  - Data Load
  - Workflow
  - Share
  - CMIS
- Alfresco Server Compatibility
- Running the Out-of-the-box tests with Maven
  - Prerequisites
    - Java
    - MongoDB
    - Apache Maven
  - Starting the Benchmark Management Server
  - Running the Sample test suite
  - Running the Sign-Up test suite
  - Running the CMIS test suite
  - Running the Data Load test suite
  - Running the Share test suite
- Getting Started Writing Your Own Tests
  - Pre-requisites
  - Introduction
  - Generate a sample load test project based on Maven Archetype
  - The sample load test project structure
  - Replacing the sample load test with a Web Script load test
    - Update the Sample Event Processors so they do the Web Script Test
    - Implementing the event DAO and DTO
    - Defining the Spring test context
    - Defining the test properties

Running the Web Script Invocation Load Test
Load Testing using Stand-alone Servers
References

Introduction

If you are an Alfresco developer there will come a time when you need to load and stress test your Alfresco solution. This is usually to make sure a Service Level Agreement (SLA) can be honored, the system performs as per customer Non Functional Requirements (NFRs), or just as a standard procedure to keep a high quality on what is delivered, and be sure the system will not break down when all the users start loading content into it.

The question is then, how to go about this? It might seem like a daunting task at first. You are not sure what tools to use, what APIs to use, if there is already something built and available, how long is it going to take etc. This is where the Alfresco Benchmark Toolkit is going to come in handy and help you out. It is purpose built by Alfresco for just this task.

The Alfresco Benchmark Toolkit provides the following:

- **Management Server** for creating tests, scheduling tests, and downloading test results
- **Driver Server(s)** for executing tests (doing the actual work and calling the Alfresco servers)
- **A framework** for writing load/stress tests
- A list of **out-of-the-box tests** that are ready to use

The Alfresco Benchmark Toolkit is applicable to a number of scenarios:

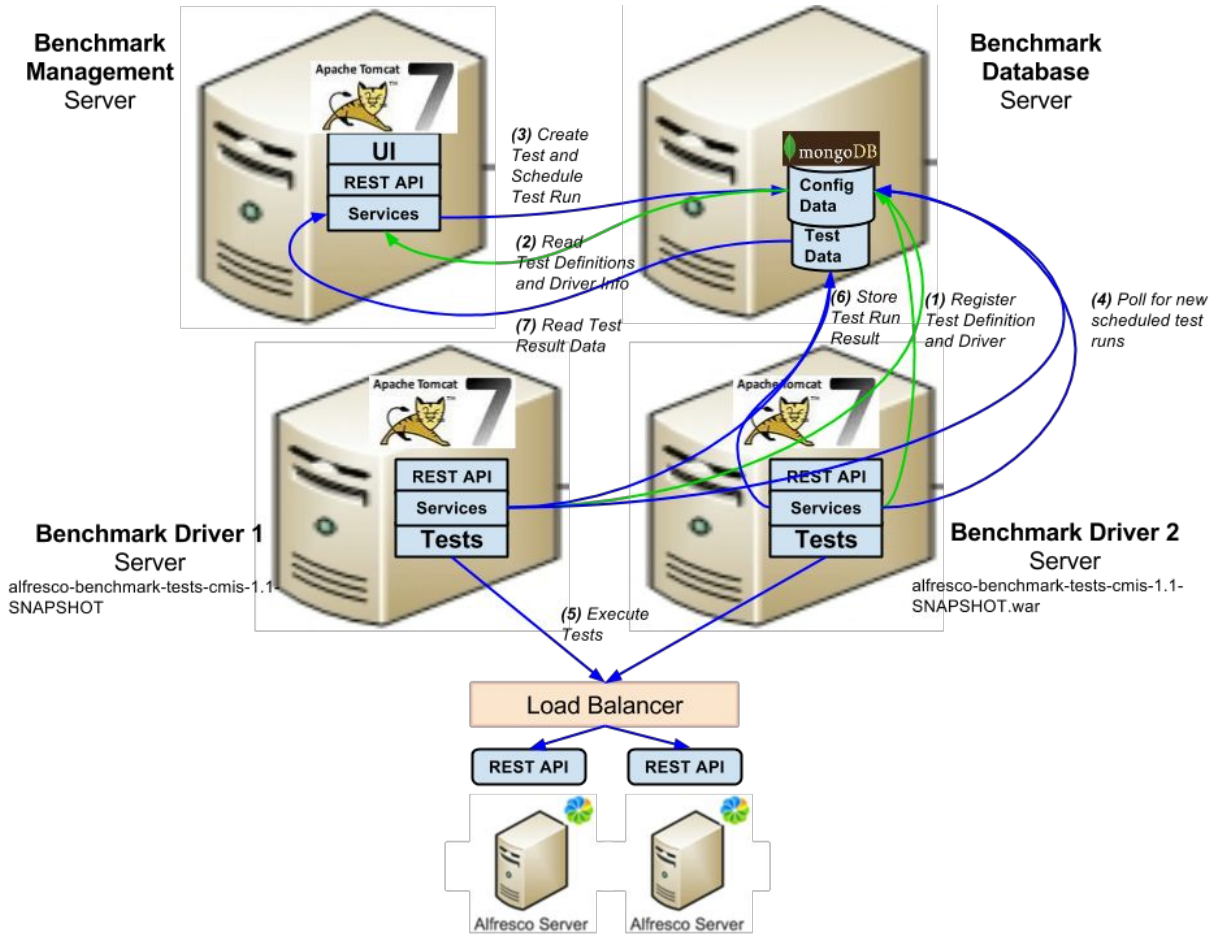
- **Load & Stress Testing** - this is the main purpose of the toolkit and it can be used to build custom benchmarking solutions for a specific Alfresco implementation.
- **Performance Testing** - if you have the need to test performance of APIs from remote office locations, then you can deploy Driver servers in these locations and see how that affects the test results.
- **Bulk Data Loading** - the framework can be used to populate your repository with massive amounts of content and users, if you have a specific need to do that and don't know how to go about it.

Architecture

The basic distributed load test setup with the Alfresco Benchmark Toolkit might look similar to something you would see when using Apache JMeter. You have a management server where you create and configure tests, and then you create test runs that are executed by driver servers on different nodes.

There is however a major difference between an Alfresco Benchmark Toolkit setup and a JMeter setup. The JMeter setup is a Master-Slave setup where the Master server tells the slave servers what to-do, and needs to know about the slave servers. On the other hand, the Alfresco Benchmark toolkit setup has no Master-Slave concept, it works on fully decoupled servers: the Management server has no knowledge of the Driver servers and vice versa. The only central point of communication in a Benchmark Toolkit distributed testing setup is the database, which every server uses as a communication broker.

The following picture illustrates a typical setup with the Alfresco Benchmark Toolkit:



So the way it works is that it starts with the Driver servers, which have been configured with one or more load test suites that they can run. In the above example configuration, each one of the Driver servers have been started with the CMIS 1.1 test suite deployed (this is just a standard WAR) to an Apache Tomcat 7 instance. A Driver server can have many different load test suites (WARs) deployed, even different versions of the same test suite.

When the Driver servers start up they will register what load test suites they can run in the database **(1)**. The Management server can then read the registered test suite definitions **(2),**

and a tester can create tests and test runs based on those **(3)**. The test runs are then scheduled by the tester **(3)**. The Driver servers will see the scheduled test runs and start the tests, more around this later on **(4)**.

After the Driver servers have started the tests they will begin calling the Alfresco servers, performing the actual testing **(5)**. When a Driver server is finished with executing a test it logs the result in the database **(6)**. The tester can then read the test results via the Management Server **(7)**.

Concepts

As you can imagine, there are a couple of concepts/components that are good to get up to speed on before we start using the Alfresco Benchmark Framework. The following section goes through the most important ones.

Events

A load test is composed of one or more events. An event is just a name of something that happens during the load test. All tests begin with an event called **start**, which is unique within one load test and is hard-coded into the system. An event can be scheduled, which means that it should happen at a specific time in the future. If an event is not scheduled then it happens at current time (immediately).

Events can carry data with them to be used during processing of the event. For example, if you have an event called **createFolder**, then the data would contain properties such as `folderName`, `session`, `parentFolder` etc. All this data is persisted in MongoDB from where it can be fetched when it is time to process the event.

There is also the possibility to use transient data that is only applicable to a sequence of events for a specific load driver. This data is kept in memory (i.e. not stored in MongoDB) and only available during the run of the test.

Data can also be attached to sessions with the session ID being carried along through a sequence of events.

All timings provided in the test results are about how long it took to process an event. Events can be handled with success or failure.

For more information see:

https://github.com/derekhulley/alfresco-benchmark/blob/master/server/src/main/java/org/alfresco/bm/event/Event.java

https://github.com/derekhulley/alfresco-benchmark/blob/V2.0.3/server/src/main/java/org/alfresco/bm/session/SessionService.java#L29

Event Processors

Defining events does not mean that things will start happening in the load test. By defining an event we tell the system what we want to happen, and what data is expected to be available, but not how it should be done. The how part is where the Event Processors come into the picture, they do the real test work.

For each event that is defined there also need to be an associated event processor that implements how that event should be processed during a load test run. The Event Processor basically implements a specific load test step, if you think of a load test as built up of multiple individual sequential steps.

The Event Processor is also where the timing of the event happens, which is a really important part for a load test. The timing of the events happens automatically as all Event Processors that you implement will extend an abstract Event Processor implementation that has a stopwatch built in. The only thing we need to do when implementing a processor is to stop and resume the stopwatch at appropriate places. For example, some processing, such as verifying and extracting event data, should not really be included in the timing.

At the end of an Event Processor implementation we need to set up what the result was of processing the event, which will be stored in MongoDB, and what the next event(s) is that we want to process as part of the load test.

For more information see:

https://github.com/derekhulley/alfresco-benchmark/blob/master/server/src/main/java/org/alfresco/bm/event/EventProcessor.java

Event Results

As mentioned above, an Event Processor should return an Event Result. This allows you to specify if the processing of the event succeeded or failed. It also allows you to store additional data in MongoDB, to be retrieved afterwards.

An Event Result also contains a list of next events to be processed. This allows you to trigger next Event Processors or Event Producers.

For more information see:

https://github.com/derekhulley/alfresco-benchmark/blob/master/server/src/main/java/org/alfresco/bm/event/EventResult.java

Event Producers

Event Producers also take incoming events, but they should not do any real test work. Event Producers are not timed, and no results are recorded. They can be used to redirect an event to another event with a delay, or anything else related to the test setup, not to do the test work.

There are a number of Event Producers available out of the box for you to use:

- `RandomRedirectEventProducer` - redirects events randomly based on relative weights
- `RedirectEventProducer` - Redirects the inbound event into a single, renamed, possibly-delayed event
- `TerminateEventProducer` - Typically used as last step of a load test to stop/terminate it. Always produces nothing. This is useful to cheaply destroy events

For more information see:

https://github.com/derekhulley/alfresco-benchmark/blob/master/server/src/main/java/org/alfresco/bm/event/producer/EventProducer.java

Data Mirrors

Data mirrors are collections in MongoDB that provide the load drivers with long-lived access to data that would not easily be obtained when a test is run. Very often, a server-in-test will be alive for weeks, being hit with all manner of load tests and investigations. During this process, the server will retain much of its state, such as the user base created. When this data is required by a test run, it is not feasible to rebuild the data using the test server because it could take a long time and is not really of any interest to the tester.

By allowing test implementations access to MongoDB directly, they are able to store any data states that might be useful for the life of the server-in-test. Additionally, test runs can use a mirror to prevent attempts to create duplicate data and avoid having to use naming conventions to generate any long-lived data. More about Data Mirrors when we go through the out-of-the-box tests.

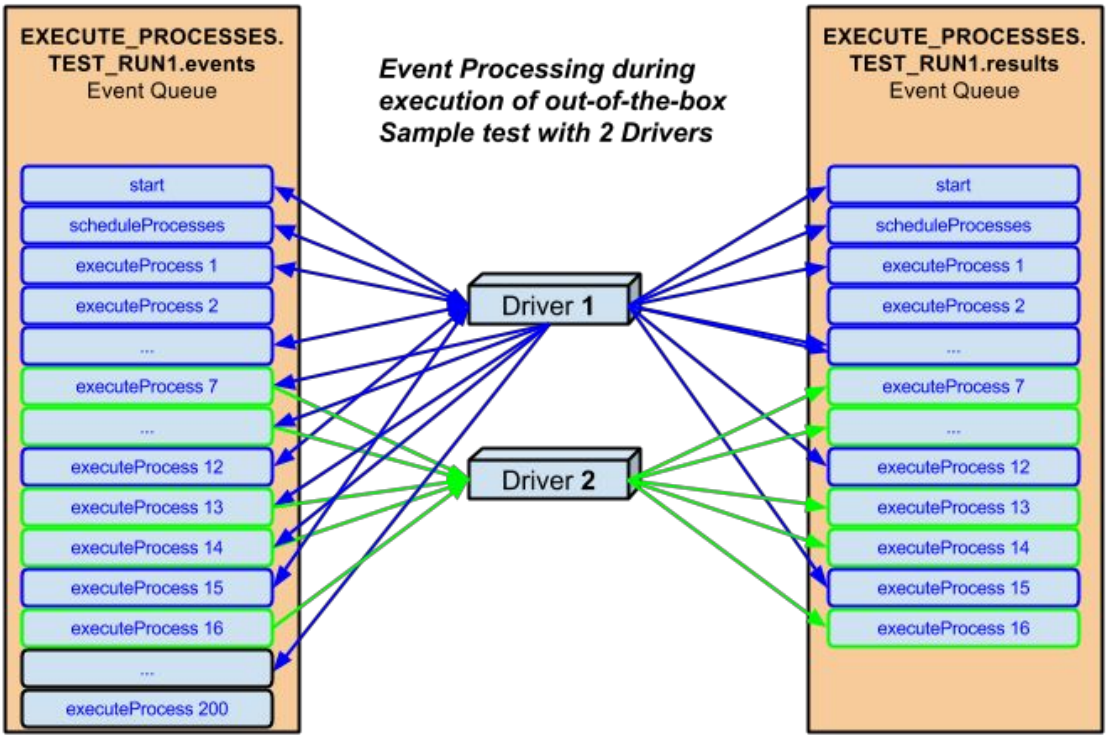
Driver Event Processing

So far we have just touched the surface for how the load testing actually works under the hood. In this section we will take a closer look at how the event processing works when the load test Drivers start and when a load test is kicked off.

A test suite is built up of a number of events that can be executed by any Driver server that has this test suite deployed. All the events are registered on an event queue by a Driver before they are processed by a Driver. This makes it possible for any Driver to jump in, grab, and process events from the event Queue. This means that the more drivers you add, the more events can

be processed in parallel. And the events can be processed on different types of hardware and network connections depending on where the Drivers have been deployed.

We are going to start looking at the event processing in the context of the out-of-the-box Sample test. This test was originally created to simulate testing workflow implementations, so it has events called `scheduleProcesses` and `executeProcess`. The following picture shows an example test run with the Sample test:



As we have learned previously, a test has one and only one `start` event. This event is registered in the event queue by one Driver, which is Driver 1 in the above picture. Both drivers compete to do this but only one Driver can change the state of the test from `scheduled` to `started`.

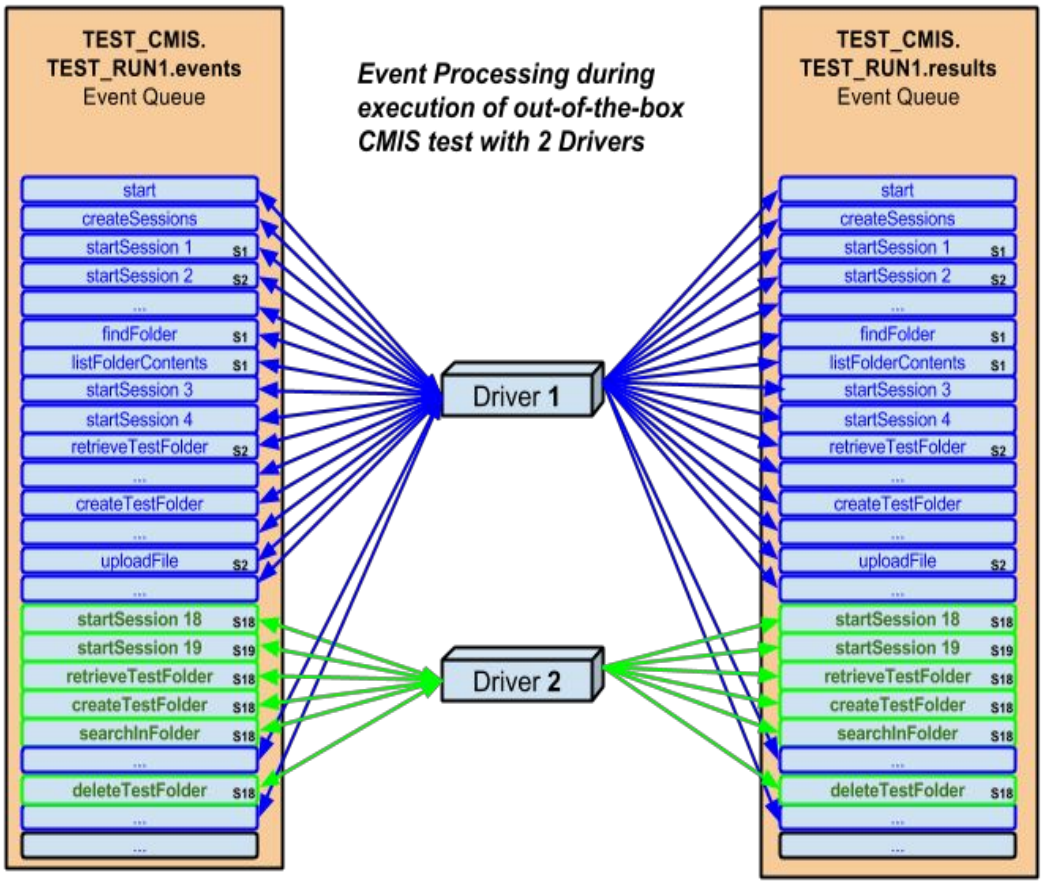
Driver 1 will lock the start event so it cannot be processed by some other Driver, and then process it, effectively starting the test. Driver 1 then registers the `scheduleProcesses` event, and immediately grabs and locks the event for processing. Processing the `scheduleProcesses` event results in 200 `executeProcess` events being put onto the event queue by Driver 1.

So far Driver 2 has not been involved as it has not had a chance to grab and lock any events in the queue. However, when Driver 1 is busy executing process 1-6, Driver 2 manages to grab and execute process 7 - 11. And then it continues with each driver grabbing some

`executeProcess` events until all of them have been processed. When an event has been processed the result of that is stored in the Event Results collection. So it is easy to see after a test has been run what drivers executed what events. The Event Result also contains execution time and contained data.

So you might be wondering, how can Driver 2 get any chance to actually execute anything at all? Why would not Driver 1 just grab all events in one go and execute them? Events are nominally assigned to random live drivers for execution; for a short time, only the assigned driver can grab the event; once the time expires, the event can be grabbed by any available driver. This is specifically useful if Driver 2 would be running in a remote location with low bandwidth connection into the database.

In this scenario we have assumed that there is no in-memory session data that is needed by multiple sequential events. When this is the case, such as when running the out-of-the-box CMIS Test suite, it would not work for multiple Drivers to just grab an event if the session data for it is in some other Driver's memory. The following picture illustrates the result of running the CMIS Test suite with 2 Drivers:



So here we can see that Driver 1 kicks things off again and creates a number of sessions that will contain CMIS session data looking something like this:

```
{
        "_id" : ObjectId("5564749644ae186e9b5a32e8"),
        "startTime" : NumberLong(1432646806592),
        "endTime" : NumberLong(1432646816665),
        "data" : {
        "repository" : "Repository Info [id=-default-, name=, description=,
capabilities=Repository Capabilities [all versions searchable=false, capability ACL=MANAGE,
capability
…
version supported=1.1, extension features=null][extensions=null]",
        "user" : "0000173.Test@00001.example.com"
        }
}
```


This event data is a non-serializable CMIS session and exists only in the memory of Driver 1, which will complete processing of all events for a test associated with one of these CMIS sessions. We can see for example that Session 1 is started by Driver 1 and then it processes event `findFolder` and `listFolderContent` sequentially. Driver 2 will also get a chance to participate in the testing but it will also create CMIS sessions and execute all events for a test sequentially.

So when in-memory event data is needed it is important to know that all events using that data will be executed by the same Driver. In-memory event data is automatically detected but can specifically be requested by an EventProcessor, if required.

The event queues that you have seen above are available in the MongoDB database as collections. These collections are named with the following convention: `{Test Definition Name}.{Test Run Name}.events` and `{Test Definition Name}.{Test Run Name}.result`. If you look in the .events collection you will not find anything if the test has completed. However, the .result collection contains all the information for when events were executed. When sessions are involved you will also have a collection called `{Test Definition Name}.{Test Run Name}.session`.

MongoDB Databases

Note that test configuration data and test execution data are stored in separate databases. This is useful if you work with a customer, and the test data is sensitive, and you want more secure storage for it than the test configuration.

Here is an example of the two databases:

Test Library (out-of-the-box tests)

As mentioned in the introduction, there are tests available for you to use, so you don't have to start from scratch when you are trying out the framework. The following list explains the different kinds of tests that are available:

Sample

When you download the Benchmark Framework source code you also get a simple test for demonstration purpose. It simulates starting a number of workflow processes. It does not require an Alfresco server to be running. For more information about this test see the Event Processing section above.

Sign Up

Logging into Alfresco at the start of a test run is probably mandatory for most kinds of tests. You can actually not do very much with the Alfresco Repository content if you are not signed in as an Alfresco user. So what this means is that you need tests that can populate the repository with the number of users that your solution should support. So if you have requirements that the Alfresco solution should support 400 casual users, then the Sign Up tests can help you create these 400 test users, and have them ready to be used by other tests.

When a user is created in Alfresco the user's details are also mirrored in the MongoDB database for reuse by subsequent tests.

For these tests you only need an Alfresco Repository instance running (i.e. alfresco.war), the Share web application (share.war) is not needed.

For more information:
https://wiki.alfresco.com/wiki/Running_Benchmark_Applications:_Alfresco_Sign_Up

Data Load

Creating Alfresco Share sites and uploading content to their Document Libraries are very common tasks in any Alfresco solution. This group of tests can be used to create loads of sites with folders and files in their Document Libraries. You can specify the number of sites you want, how many sub-folders each site's Document Library should have, how many files you want to be uploaded to each folder, and finally the folder depth in the Document Library. The Data Load tests can also be used to create a Records Management (RM) site with users.

For these tests you only need an Alfresco Repository instance running (i.e. alfresco.war), the Share web application (share.war) is not needed.

These tests assumes that you have already populated the repository with users by using the Sign Up tests.

For more information:
https://wiki.alfresco.com/wiki/Running_Benchmark_Applications:_Alfresco_Data_Load

Workflow

The majority of Alfresco solutions that are implemented today include one or more workflows that should process Repository content in one way or another. To be able to simulate the number of workflow instances that you need to support, based on the customer requirements, you can use these out-of-the-box Workflow tests.
They can start and progress Activiti workflows via the Alfresco Workflow API. The tests will record how long it takes to progress through the different parts of the workflow and if any errors occur.

For these tests you only need an Alfresco Repository instance running (i.e. alfresco.war), the Share web application (share.war) is not needed.

These tests assumes that you have already populated the repository with users by using the Sign Up tests.

For more information:
https://wiki.alfresco.com/wiki/Running_Benchmark_Applications:_Alfresco_Workflow

Share

So far the out-of-the-box tests have not covered the User Interface (UI), instead they have focused more on loading vast amounts of data into the Repository. What about testing UI stuff in Share, how would you go about that? The Share tests is using Selenium-based testing with Share Page Objects (PO), and shows you many different examples of how you can build tests that are UI focused.

These tests assumes that you have already populated the repository with users by using the Sign Up tests. It is not necessary to have pre-created sites but the Share tests will work against any existing Share data.

CMIS

The out-of-the-box tests so far requires Alfresco as a CMS server. If you would like to use the Alfresco Benchmark toolkit to test some other CMS server, then you could do that with the CMIS test suite as most CMS servers supports the CMIS standard. The CMIS tests cover the basic CRUD operations that you can do with the CMIS API, and there is also a template for how to build use-case specific tests based on your specific requirements.

These tests assumes that you have already populated the repository with users by using the Sign Up tests. However, the Sign Up tests will only populate an Alfresco server with users as there is no API in the CMIS standard for creating users. If you are not using Alfresco then you need to create users in the target CMS server and then import them into the MongoDB instance.

As soon as you got the information about the users in MongoDB they can be used by the CMIS test suite.

For more information:
https://wiki.alfresco.com/wiki/Running_Benchmark_Applications:_CMIS

Alfresco Server Compatibility

When using the Benchmark Framework there are different versions of it and the associated out-of-the-box tests. The following table shows what tests are compatible with what Alfresco Server version. (see https://wiki.alfresco.com/wiki/Benchmark_Testing_with_Alfresco)

Alfresco Server Version	**Benchmark Test Versions**
Alfresco 5.1.0	Benchmark Server 2.0.9 or later *Tests:* Sign Up 2.2 or later Data Load 2.4 or later CMIS 1.2 Workflow 1.1 Public API 1.0-SNAPSHOT Share 5.1.0-SNAPSHOT (trunk)
Alfresco 5.0.0	Benchmark Server 2.0.9 or later *Tests:* Sign Up 2.2 or later Data Load 2.4 or later CMIS 1.2 Workflow 1.1 Public API 1.0-SNAPSHOT Share 5.0.x-SNAPSHOT
Alfresco 4.2.5	Benchmark Framework 2.0.9 or later *Tests:* Sign Up 2.2 or later Workflow 1.1 Share 4.2.x-SNAPSHOT
Alfresco 4.2.4	Benchmark Framework 2.0.9 or later *Tests:* Sign Up 2.2 or later Workflow 1.1 Share 4.2.4.x-SNAPSHOT

Alfresco 4.1.9	Benchmark Framework 2.0.9 or later *Tests:* Sign Up 2.2 or later Workflow 1.1 Share 4.1.x-SNAPSHOT
CMIS 1.1 compatible CMS Server	Benchmark Framework 2.0.9 or later *Tests:* CMIS 1.2

Where a specific Product Version is not listed, use the most recent version listed. For example, if you are using **Alfresco 4.2.1**, then the tests to use will be those of the latest prior release; in this example: **Alfresco 4.1.9**.

Running the Out-of-the-box tests with Maven

The following section will take you through how to run some of the out-of-the-box tests so you can get a feeling for how it all works. We will run the Manager Server and the Driver Servers directly from the Maven projects. Later on in this article we will have a look at how to use stand-alone servers.

Prerequisites

There are a few components we need before we can kick off any tests.

Java

Before you can start any form of testing with the Alfresco Benchmark Framework you need to install Java SDK version 1.7.0_51 or later on the Benchmark Management Server host and on each host running the Benchmark Driver Server. I assume you can Google and figure out how to install the JDK on your specific platform, if you don't already have it.

MongoDB

There also needs to be an instance of MongoDB, version 2.6.3 or later, running on a host somewhere. I run Ubuntu on my laptop and it includes its own MongoDB packages, but the official MongoDB packages are generally more up-to-date and has the newer versions that we need.

This is how I installed MongoDB on my Ubuntu 14.04 laptop:

```
martin@gravitonian:~$ sudo apt-key adv --keyserver
hkp://keyserver.ubuntu.com:80 --recv 7F0CEB10
martin@gravitonian:~$ echo "deb http://repo.mongodb.org/apt/ubuntu
"$(lsb_release -sc)"/mongodb-org/3.0 multiverse" | sudo tee
/etc/apt/sources.list.d/mongodb-org-3.0.list
```

```
martin@gravitonian:~$ sudo apt-get update
martin@gravitonian:~$ sudo apt-get install -y mongodb-org
```

So the above commands do the following. It starts off by importing the MongoDB public GPG Key so the Ubuntu package managers (i.e. `dpkg` and `apt`) can verify the downloaded MongoDB packages. We then tell Ubuntu where the MongoDB repository is by creating a source list file with this information. After this it is straight forward to use the standard `apt-get install` command to download and install MongoDB.

Now test the MongoDB installation by executing the `mongo` command as follows:

```
martin@gravitonian:~$ mongo
MongoDB shell version: 3.0.1
connecting to: test
```

Make sure version is newer than 2.6.3. We also need to check the logs for the port number that MongoDB uses. The Benchmark Framework servers expects the port number to be 27017. Check this via the log file as follows:

```
martin@gravitonian:~$ more /var/log/mongodb/mongod.log
2015-04-05T11:09:58.965+0100 I JOURNAL  [initandlisten] journal
dir=/var/lib/mongodb/journal
2015-04-05T11:09:58.965+0100 I JOURNAL  [initandlisten] recover : no
journal files present, no recovery needed
2015-04-05T11:09:58.988+0100 I JOURNAL  [durability] Durability
thread started
2015-04-05T11:09:58.988+0100 I JOURNAL  [journal writer] Journal
writer thread started
2015-04-05T11:09:58.997+0100 I CONTROL  [initandlisten] MongoDB
starting : pid=8446 port=27017 dbpath=/var/lib/mongodb 64-bit
host=gravitonian
```

Apache Maven

We will be building and running the Alfresco Benchmark Toolkit components, such as the Management server, from source code. Maven is used as the build tool so make sure you have Apache Maven 3 installed.

Starting the Benchmark Management Server

All the tests that we are going to run will be controlled from the same Management server, so it make sense to start it up first. This involves downloading the source code for the Benchmark Framework and building it and then starting the Management server:

```
$ git clone https://github.com/derekhulley/alfresco-benchmark
```

```
$ cd alfresco-benchmark
alfresco-benchmark$ cd server
alfresco-benchmark/server$ mvn tomcat7:run -Dmongo.config.host=localhost
```

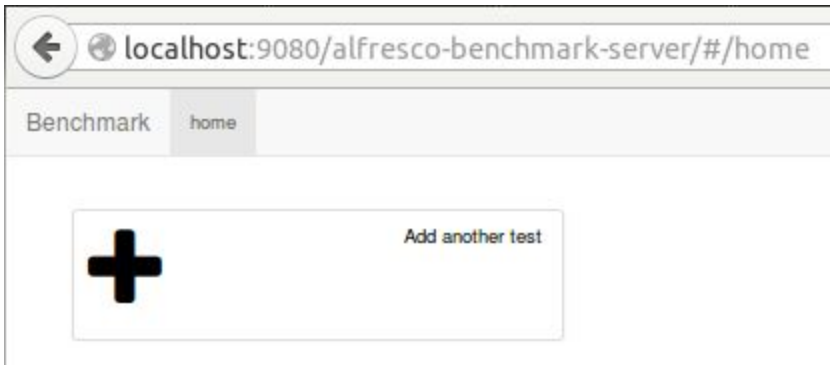
To run the Management server from maven we use the `tomcat7-maven-plugin`. It kicks off an embedded Apache Tomcat instance with the Management Server web application deployed. For more details on how the Tomcat 7 plugin is configured have a look at the Maven project file.

This is also where we make use of the MongoDB installation we did in the beginning, we configure the location of it via the `mongo.config.host` parameter. Note that this is just the location of the benchmark *configuration database* where the drivers will register themselves and the load tests that they can run. The *test data location* is configured from the Management Server UI when we create a load test.

After the Benchmark Management server has started you can access it via the following URL:

http://localhost:9080/alfresco-benchmark-server

You should see something like this if this is the first time you are running the system:



Running the Sample test suite

When you downloaded the Benchmark Toolkit to run the Management Server you also got a sample test as part of the download. This is the test that simulates starting workflow processes. This test does not actually require you to have an Alfresco installation running. The Driver servers will just execute the test events in memory and not call any external applications (for more information about this test see the Driver Event Processing session above).

We will kick off 2 drivers with this test. Start Driver 1 as follows:

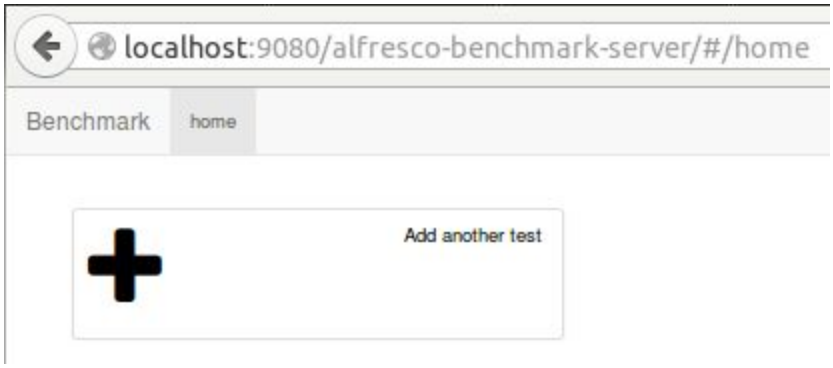
```
alfresco-benchmark$ cd sample/
alfresco-benchmark/sample$ mvn tomcat7:run -Dmongo.config.host=localhost
…
```

```
INFO: Starting ProtocolHandler ["http-bio-9081"]
```

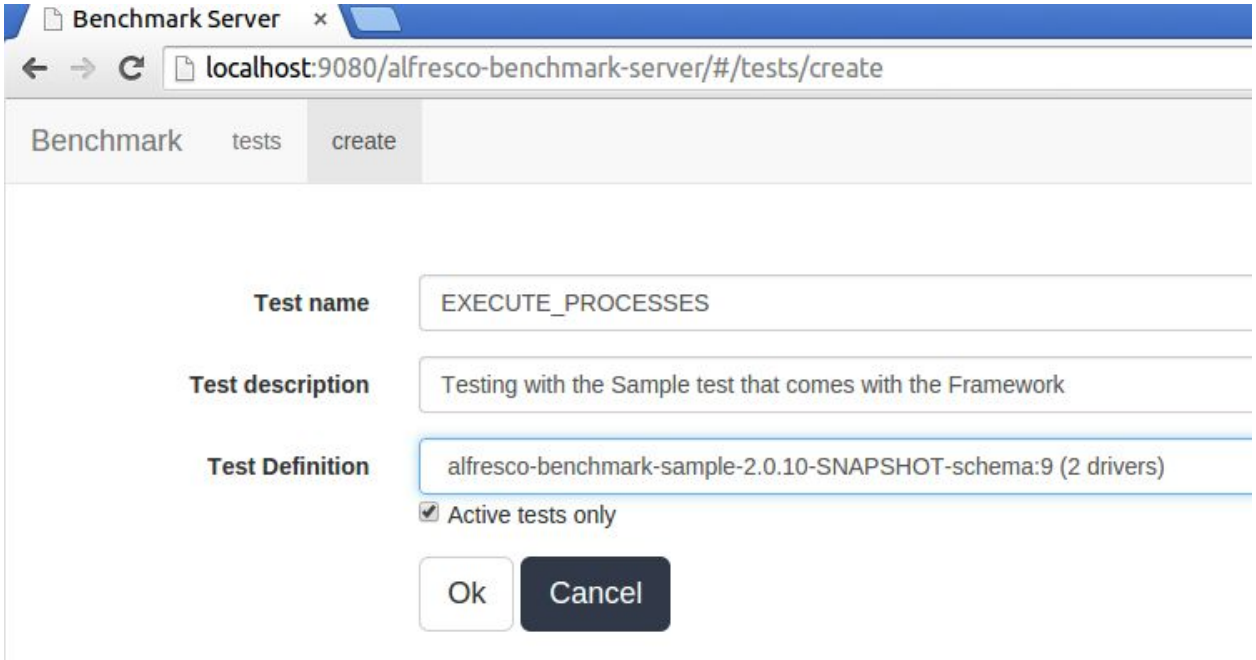
To run the Driver server from maven we again use the `tomcat7-maven-plugin`. See this project file for information about how it is configured. It kicks off an embedded Apache Tomcat instance with the Sample test suite web application deployed. To kick off a second Driver just use a different console window and supply a different port number:

```
alfresco-benchmark/sample$ mvn tomcat7:run -Dmongo.config.host=localhost
-Dbm.tomcat.port=9082
...
INFO: Starting ProtocolHandler ["http-bio-9082"]
```

We are now ready to go back to the Management server interface and create a new Sample test definition. In the following dialog, click the big **+**:

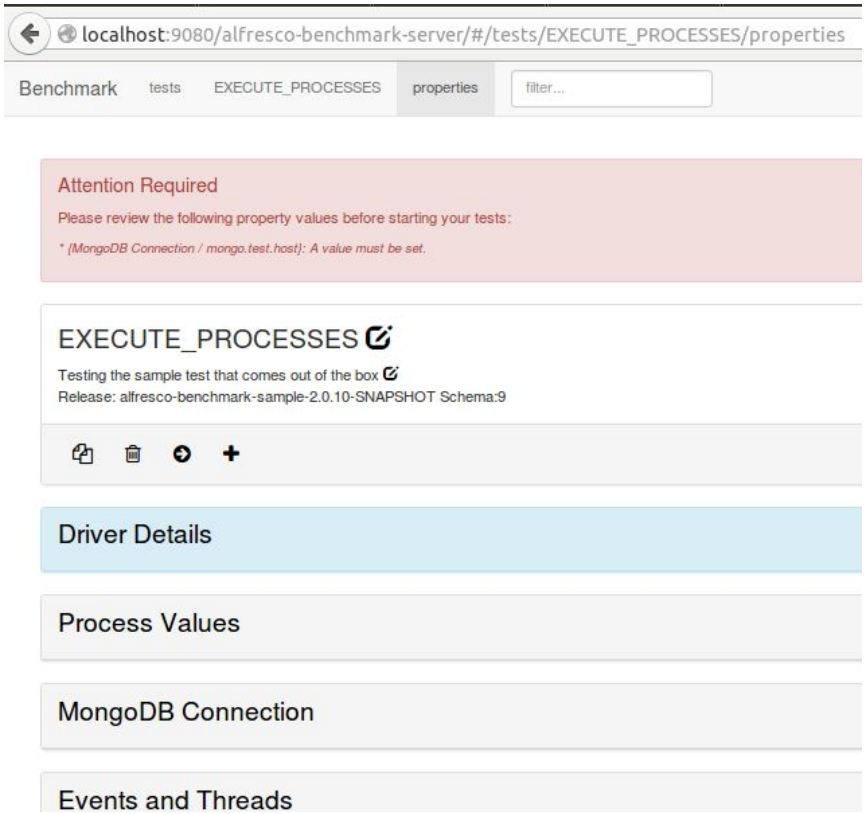


This will bring up a new dialog where we can specify a name and description for the test. After that we select which test definition (i.e. WAR) that implements it:



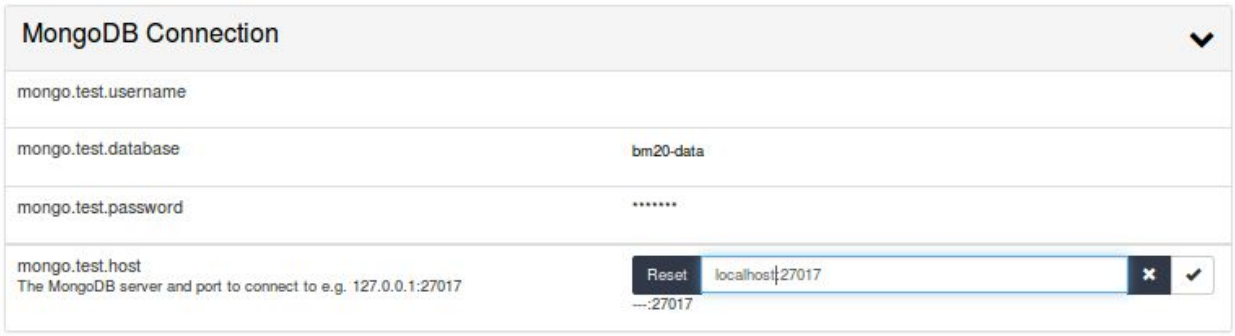
We can see that there are currently two drivers up and running that supports the sample test suite. The test definitions that you can choose from depends on the number of Benchmark Drivers that have been started, and that are connected to the same MongoDB config database as the Management Server.

Click **OK** to save this load test, you will then see the following screen:



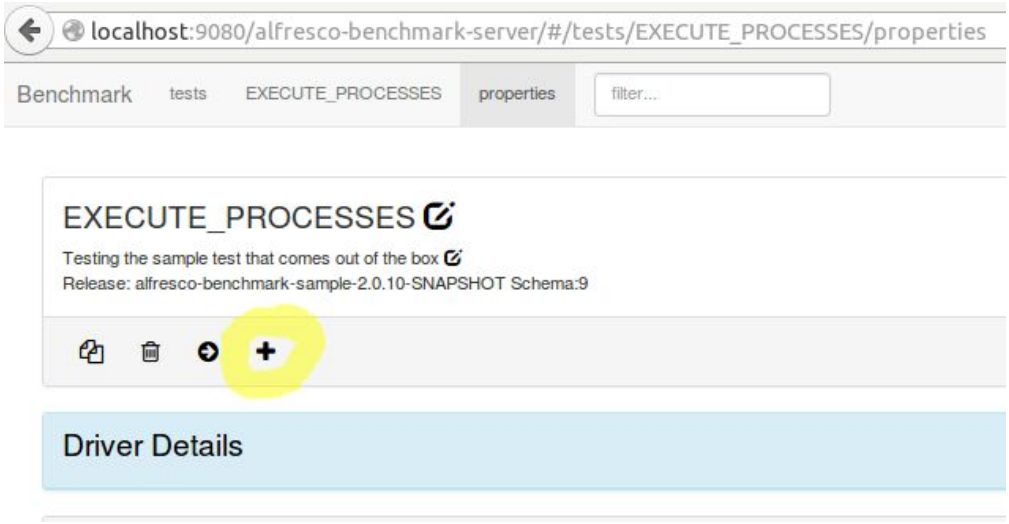
This is the configuration screen with all the properties that can be set for this test. The properties are divided into different groups, such as **Process Values** and **MongoDB Connection**. When we set these properties for the Test Definition they will be used as defaults for each Test Run that we create later on. But it is also possible to override a property value for a specific Test Run. As we will see in this article, a lot of these property groups are reused in many of the tests, and you can reuse them when you write your own new custom tests.

Here we can also see that there is red warning message at the top of the screen. It is telling us that we need to configure a location for where all the Test Data should end up when we run tests, it can be a different MongoDB instance if we wanted to (in case we got sensitive client data for example). The IP address or server name needs to be durable and recognisable by all drivers. In our case we are going to use the same local MongoDB instance. Click the **MongoDB Connection** section to set this up:

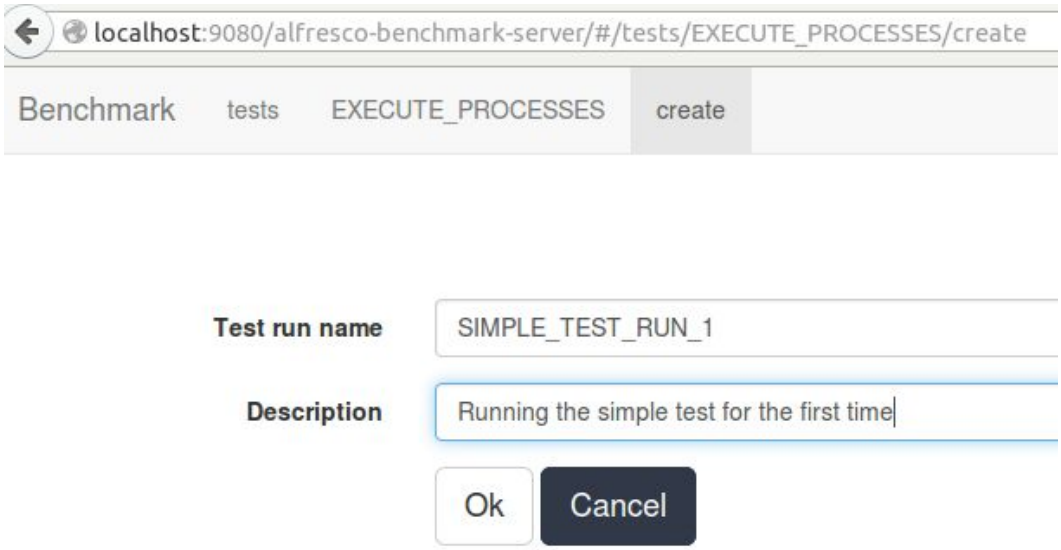


There are a number of other properties sections where we can configure stuff, most are standard and look the same for all load tests implementations, but there are also sections specific to the load test implementation, such as the **Process Values** one in our case:

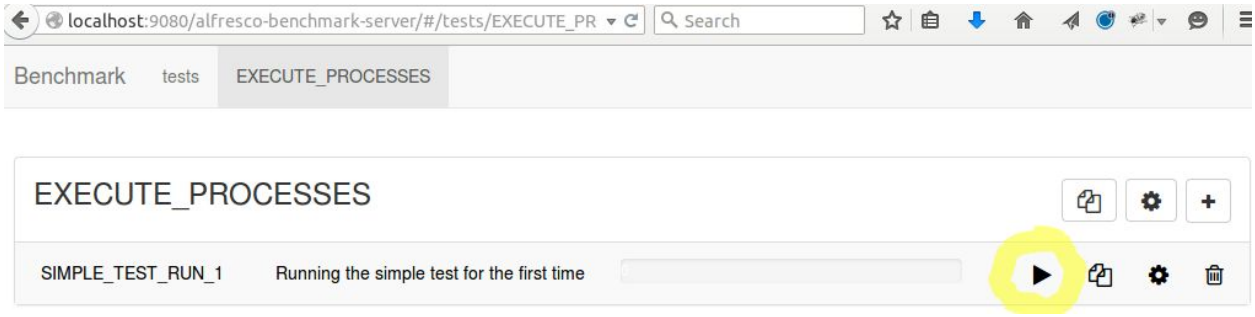
This is where we can configure the characteristics for the Sample test definition. We could for example have two different Sample load test definitions, one with process count 200 and one with process count 1000.
Now to run a test based on this load test configuration we need to create a test run. We do this by clicking on the **+** at the top of the screen, just under the summary and description of the test:



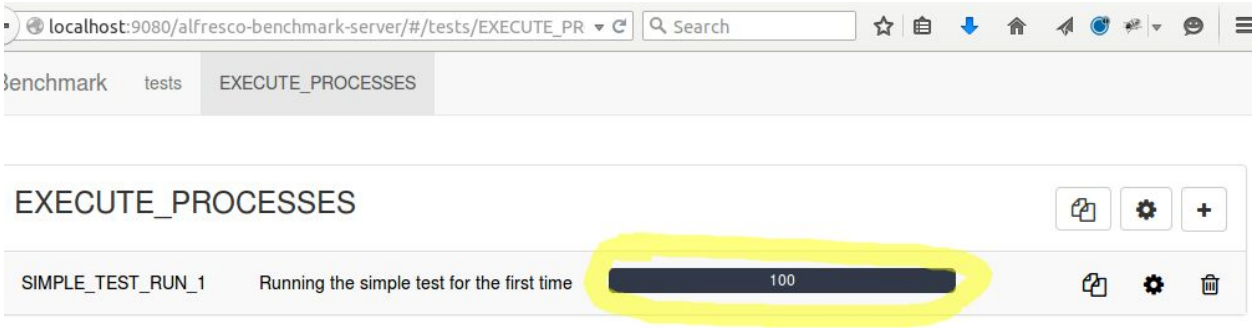
This presents the following screen where we can give the test run a name and description:



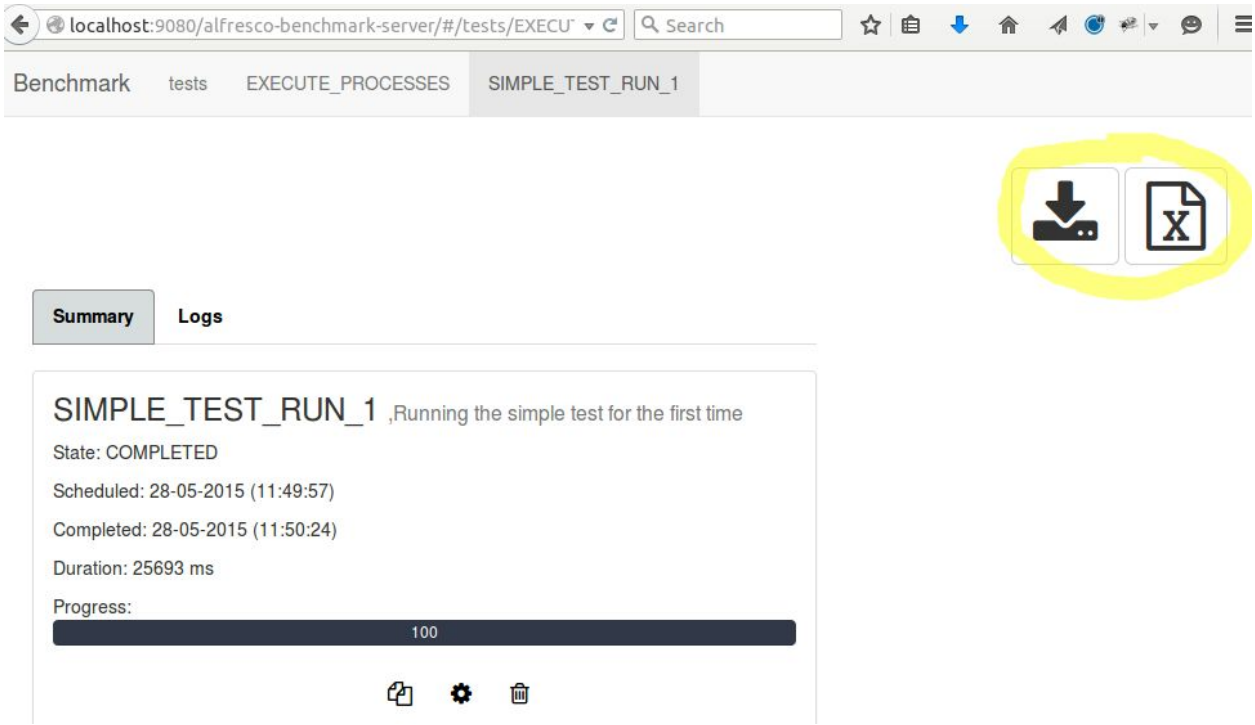
Note that the Test Definition name and the Test Run name makes up the name of the MongoDB collection where the Test Data will be stored, in this case `bm20-data/Collections/EXECUTE_PROCESSES.SIMPLE_TEST_RUN_1.*`. Click **OK** and you will see the screen where the test runs can be started:



Click the arrow to start the load test, it will be executed by both of the Benchmark Drivers that we started. When the test is finished you will see the following screen:



Here you can click on the test row to navigate to a screen where you can download the load test result as a CSV or Excel file:



This completes the whole process of running the Sample load test from the Benchmark Management server.

Running the Sign-Up test suite

Another test suite that is available out of the box is the Sign-Up. It will allow you to create loads of users that can be used later on by other test suites, such as the CMIS test suite. In fact, some test suites, such as the CMIS one, have as a prerequisite that the Sign Up test has already been run so that there are test users available.

The Sign Up test obviously requires Alfresco to be running, so if you don't have it started, now is the time to get it up and running. If you need help with that go to the Alfresco Community download site for more information.

The Alfresco Sign Up test source code is available in a Subversion repo, use the following command to download the source code:

```
$ svn co
https://svn.alfresco.com/repos/alfresco-open-mirror/benchmark/tests/ent-signup/
tags/V2.0/ alfresco-benchmark-signup
$ cd alfresco-benchmark-signup
martin@gravitonian:~/src/alfresco-benchmark-signup$
```

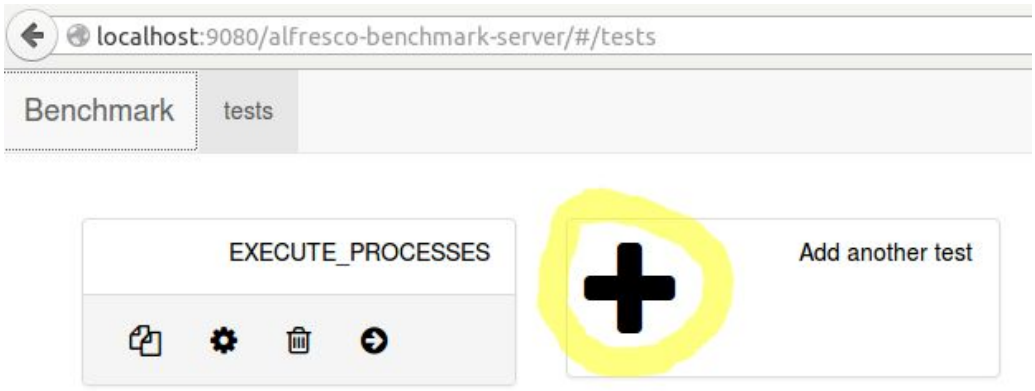
We will kick off 2 drivers with this test. Start Driver 1 as follows:

```
alfresco-benchmark-signup$ mvn tomcat7:run -Dmongo.config.host=localhost
…
INFO: Starting ProtocolHandler ["http-bio-9082"]
```

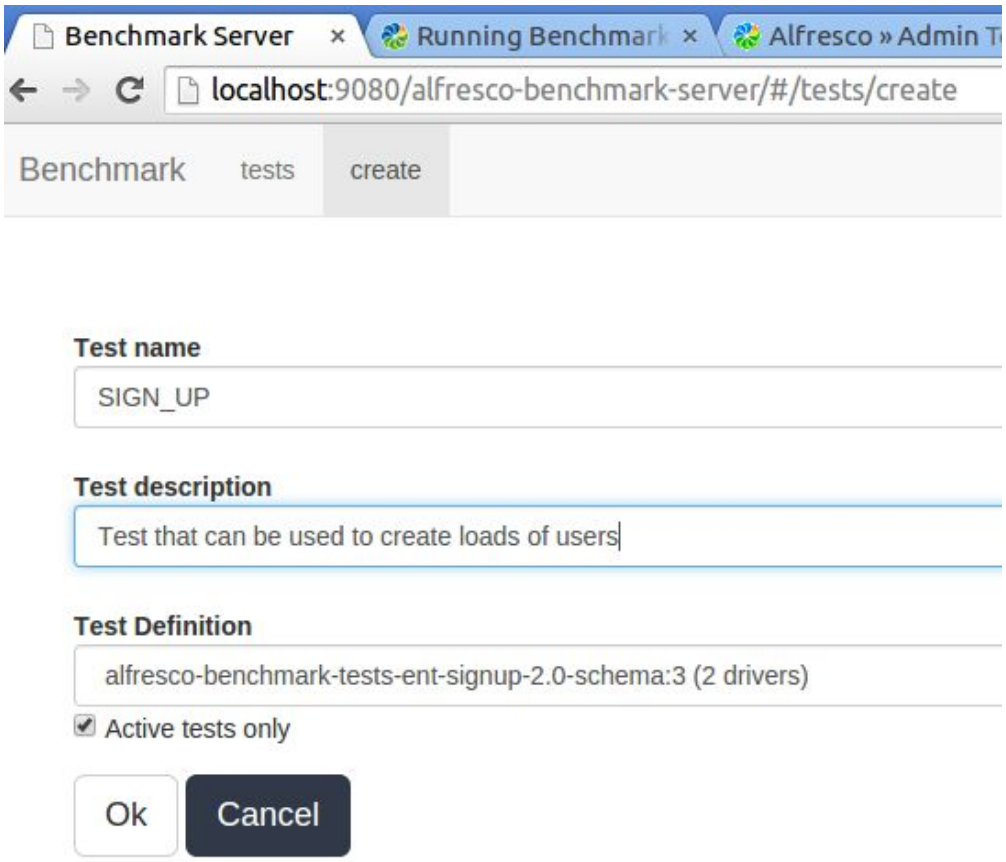
To run the Driver server from maven we again use the `tomcat7-maven-plugin`. For more information about the Tomcat plugin configuration see the project file. It kicks off an embedded Apache Tomcat instance with the Sign Up test suite web application deployed. To kick off a second Driver just use a different console window and supply a different port number:

```
alfresco-benchmark-signup$ mvn tomcat7:run -Dmongo.config.host=localhost
-Dbm.tomcat.port=9083
...
INFO: Starting ProtocolHandler ["http-bio-9083"]
```

We are now ready to go back to the Management server interface and create a new Sign Up test definition. In the following dialog, click the big **+**:



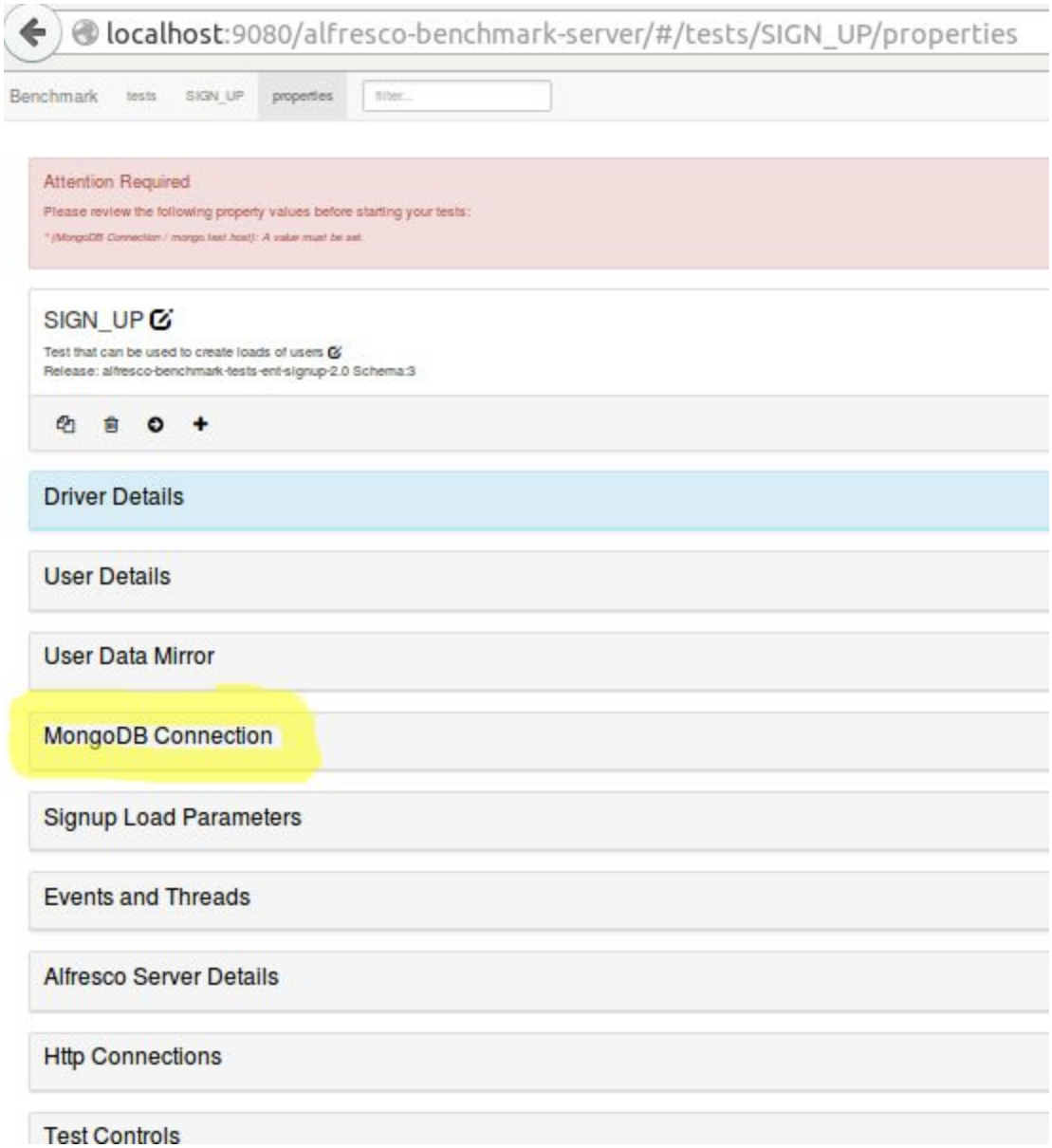
This will bring up a new dialog where we can specify a name and description for the test. After that we select which test definition (i.e. WAR) that implements it:



We can see that there are currently two drivers up and running that supports the Sign-Up test suite. The test definitions that you can choose from depends on the number of Benchmark Drivers that have been started, and that are connected to the same MongoDB config database as the Management Server.

However, there is also the “**Active tests only**” checkbox that will have a say in what definitions that are available in the drop down. Uncheck it and you will see all test definitions that was once active with Drivers, and you can create a test for anyone of them, even if there are no active Drivers (Try it and you should see the sample test popup).

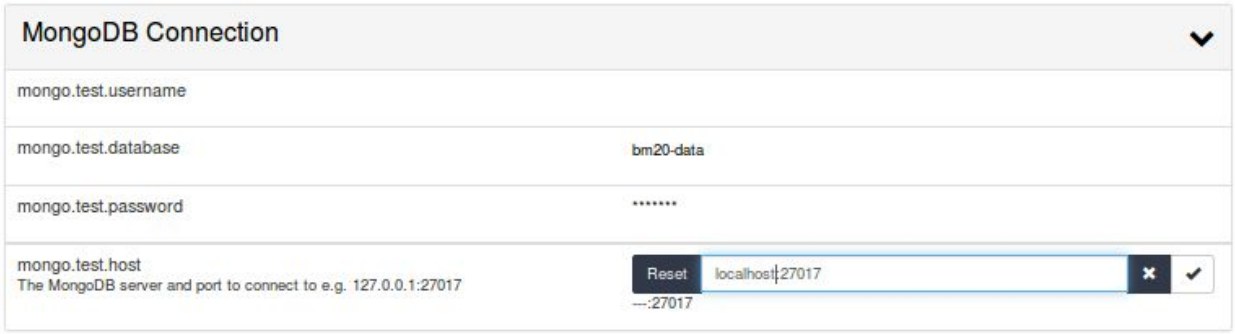
Click **OK** to save this load test, you will then see the following screen:



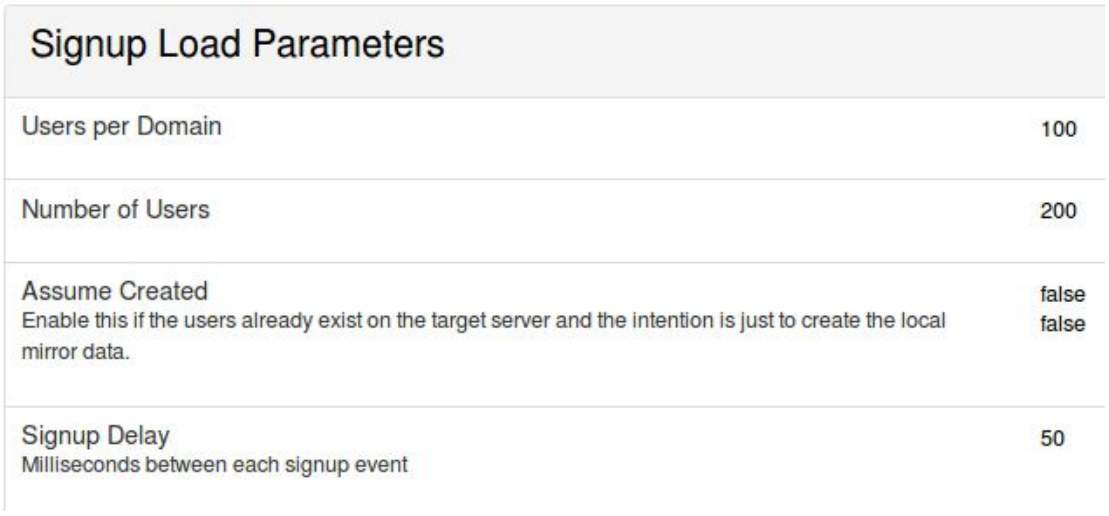
This is the configuration screen with all the properties that can be set for this test. The properties are divided into different groups, such as **Alfresco Server Details** and **MongoDB Connection**. When we set these properties for the Test Definition they will be used as defaults

for each Test Run that we create later on. But it is also possible to override a property value for a specific Test Run. As we will see in this article, a lot of these property groups are reused in many of the tests, and you can reuse them when you write your own new custom tests.

Here we can also see that there is red warning message at the top of the screen. It is telling us that we need to configure a location for where all the test data should end up when we run tests, it can be a different MongoDB instance if we wanted to (if we got sensitive client data for example). In our case we are going to use the same local MongoDB instance. Click the **MongoDB Connection** section to set this up:

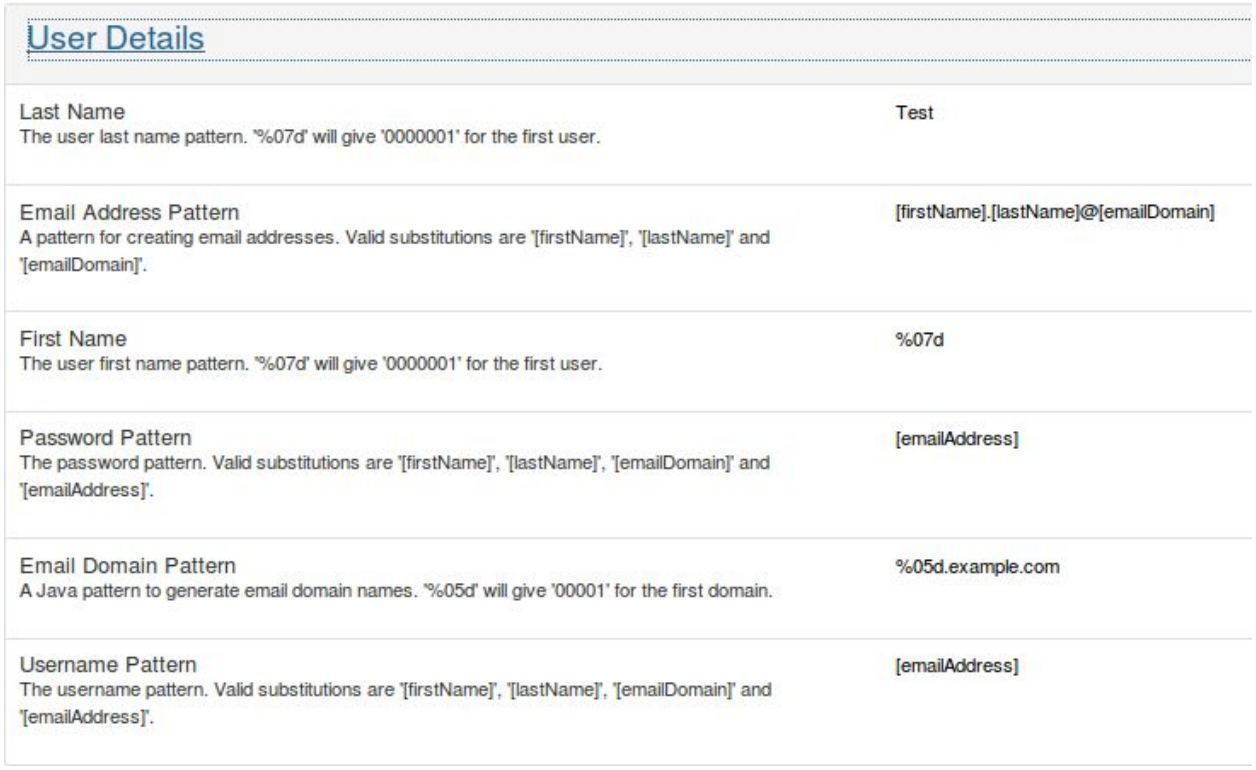


There are a number of other properties sections where we can configure stuff, most are standard and look the same for all load tests implementations, but there are also sections specific to the Sign-Up test implementation, such as the **Signup Load parameters** one in our case:



This is where we can configure the characteristics for this test definition. We could for example have two different Sign-Up load test definitions, one with creates 200 users and one which creates a 1000.

You might also want to control the naming of the test users, passwords, and other data around the user account, you can do this via the **User Details** group:

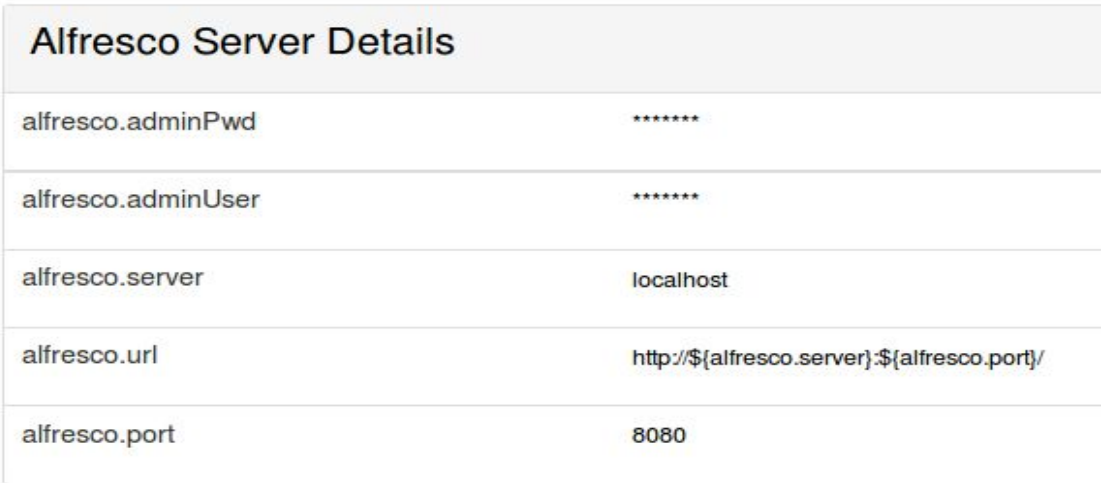


User Details

Last Name The user last name pattern. '%07d' will give '0000001' for the first user.	Test
Email Address Pattern A pattern for creating email addresses. Valid substitutions are '[firstName]', '[lastName]' and '[emailDomain]'.	[firstName].[lastName]@[emailDomain]
First Name The user first name pattern. '%07d' will give '0000001' for the first user.	%07d
Password Pattern The password pattern. Valid substitutions are '[firstName]', '[lastName]', '[emailDomain]' and '[emailAddress]'.	[emailAddress]
Email Domain Pattern A Java pattern to generate email domain names. '%05d' will give '00001' for the first domain.	%05d.example.com
Username Pattern The username pattern. Valid substitutions are '[firstName]', '[lastName]', '[emailDomain]' and '[emailAddress]'.	[emailAddress]

We are not going to change anything in the user details settings. But this configuration could be useful if you are just using the Sign-Up test to populate an Alfresco instance with loads of user accounts, and have no intention of running any other tests, but will use the accounts for something else.

We also have properties related to where the Alfresco server is running:

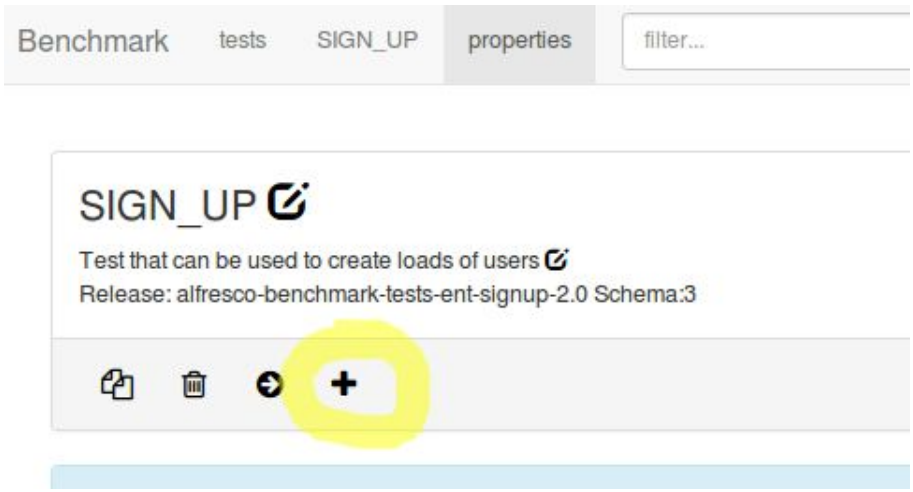


Alfresco Server Details

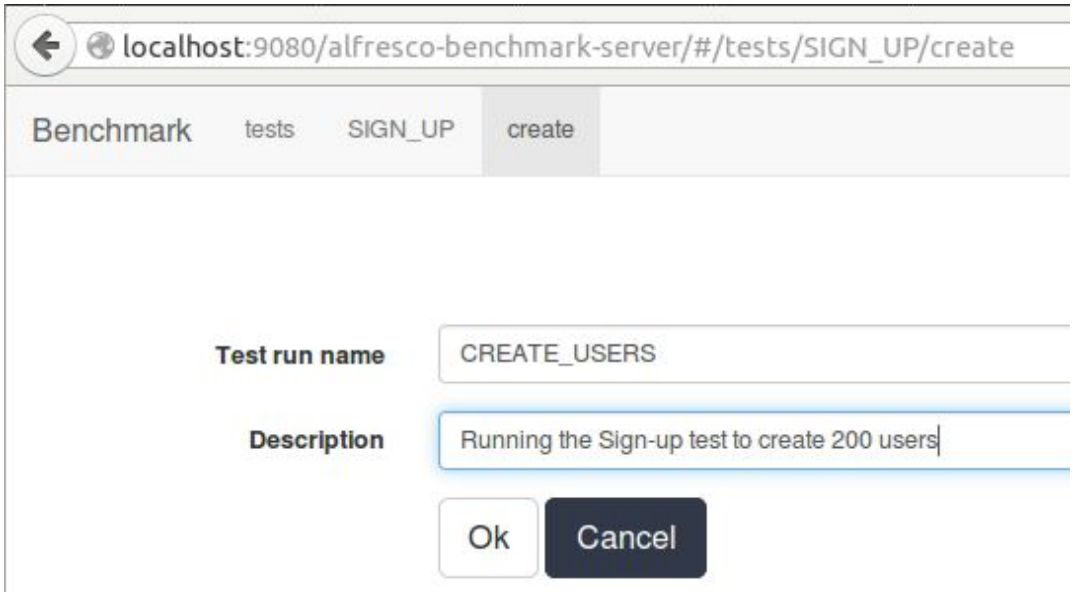
alfresco.adminPwd	*******
alfresco.adminUser	*******
alfresco.server	localhost
alfresco.url	http://${alfresco.server}:${alfresco.port}/
alfresco.port	8080

Remember that the URL needs to be accessible from all the load driver machines. In our case the default values for the Alfresco server connection and authentication is going to be fine. The username/password is set to admin/admin by default. The password field is hidden and will not appear in any logs or API results but is stored in clear text in the MongoDB database; you can configure security on the MongoDB database if this is an issue.

Now to run a test based on this load test configuration we need to create a test run. We do this by clicking on the **+** at the top of the screen, just under the summary and description of the test:

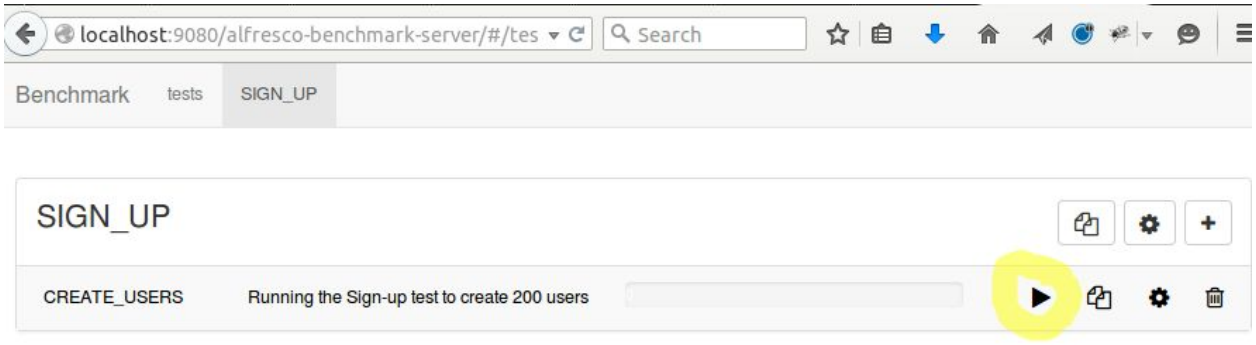


This presents the following screen where we can give the test run a name and description:

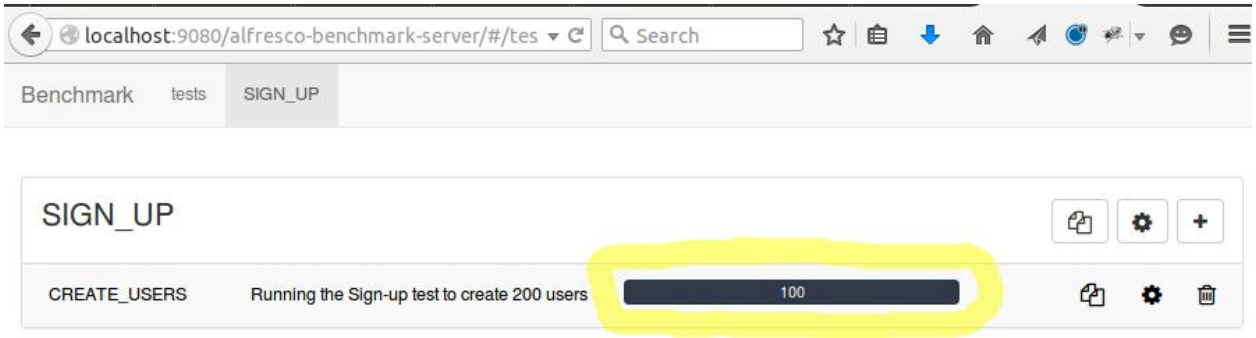


Note that the Test Definition name and the Test Run name makes up the name of the MongoDB collection where the Test Data will be stored, in this case `bm20-data/Collections/`

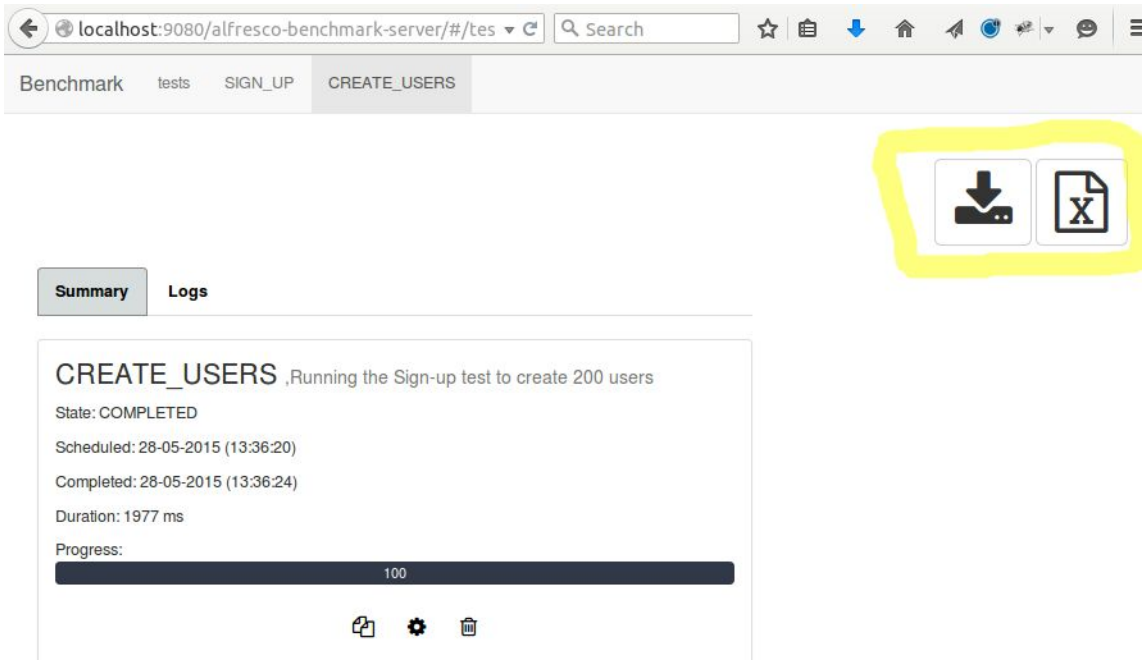
`SIGN_UP.CREATE_USERS.*`. Click **OK** and you will see the screen where the test runs can be started:



Click the arrow to start the load test, it will be executed by both of the Benchmark Drivers that we started. When the test is finished you will see the following screen:

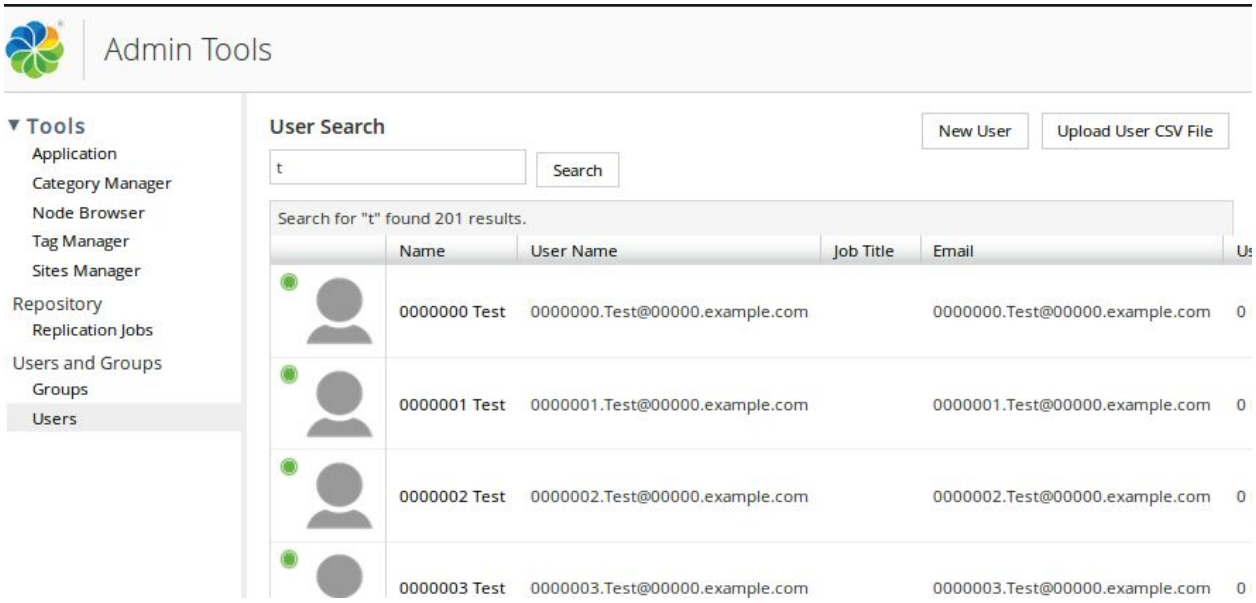


Here you can click on the test row to navigate to a screen where you can download the load test result as a CSV or Excel file:



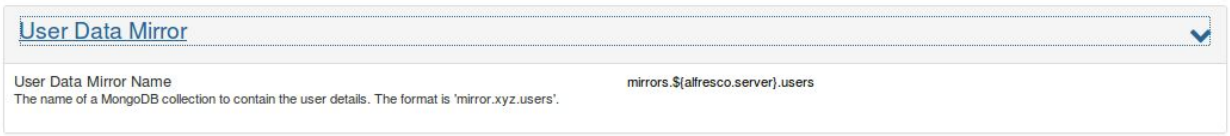
This completes the whole process of running the Sign-UP load test from the Benchmark Management server.

Now, to make sure that the users have actually been created, login to Alfresco Share and search for test users as follows:

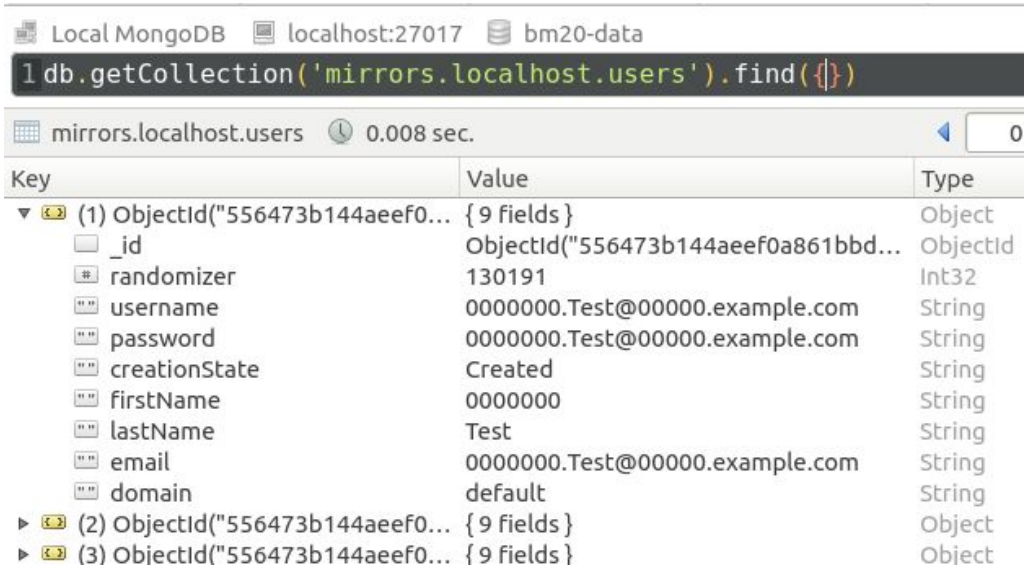


An important thing to keep in mind going forward is how the Benchmark Framework keeps track of what users that are available in the system, and that can be used by other tests, such as the CMIS test we are going to look at next. It does this by keeping a copy of the user account

information in the MongoDB database. This is referred to as so called Data Mirrors. And we got a configuration for the **User Data Mirror** that looks like this:



If you look in the database you will see something like this for the run we did:



So you are probably thinking, what happens if we run the test again, is it going to create another 200 users. No, it will see that we have already created 200 users and not do anything. If you wanted to add another 100 users you would have to create another Test Run and set the number of users to 300.

To have tests behave in an *idempotent* way is good practice, it means that you can include the Sign-Up test suite in a bigger test scenario without having to worry about if it has already been run, or if it will create a lot more users than necessary. It will also speed up the “real” tests a lot as you don’t have to wait for the users to be created every time. By “real” tests I mean for example the CMIS test suite that is dependent on the Sign-Up test suite, meaning that the Sign-Up tests are just preparation tests in this case and CMIS is what you are really testing. It is also possible to run “real” tests in parallel (e.g. Share with DataLoad and CMIS) as both of these can rely on the same user base.

Running the CMIS test suite

We now have Alfresco started and a number of test users created. We are ready to run the CMIS test suite.
The CMIS test is available on GitHub, use the following command to download the source code:

```
$ git clone https://github.com/derekhulley/benchmark-cmis.git
$ cd benchmark-cmis/
benchmark-cmis$
```

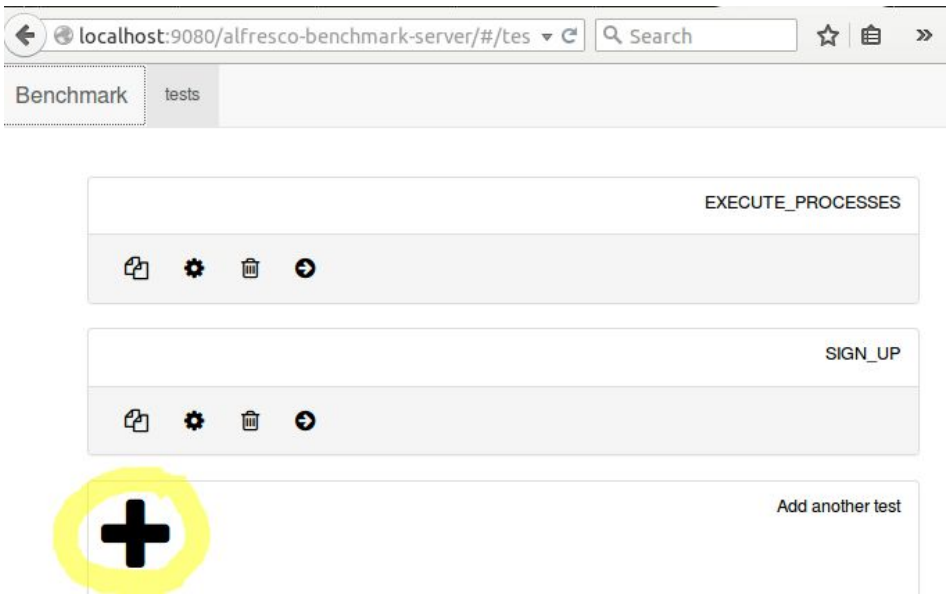
We will kick off 2 drivers with this test. Start Driver 1 as follows:

```
benchmark-cmis$ mvn tomcat7:run -Dmongo.config.host=localhost
…
INFO: Starting ProtocolHandler ["http-bio-9090"]
```

To run the Driver server from maven we again use the `tomcat7-maven-plugin`. For more information about the configuration of the Tomcat plugin see the project file. It kicks off an embedded Apache Tomcat instance with the CMIS test suite web application deployed. To kick off a second Driver just use a different console window and supply a different port number:

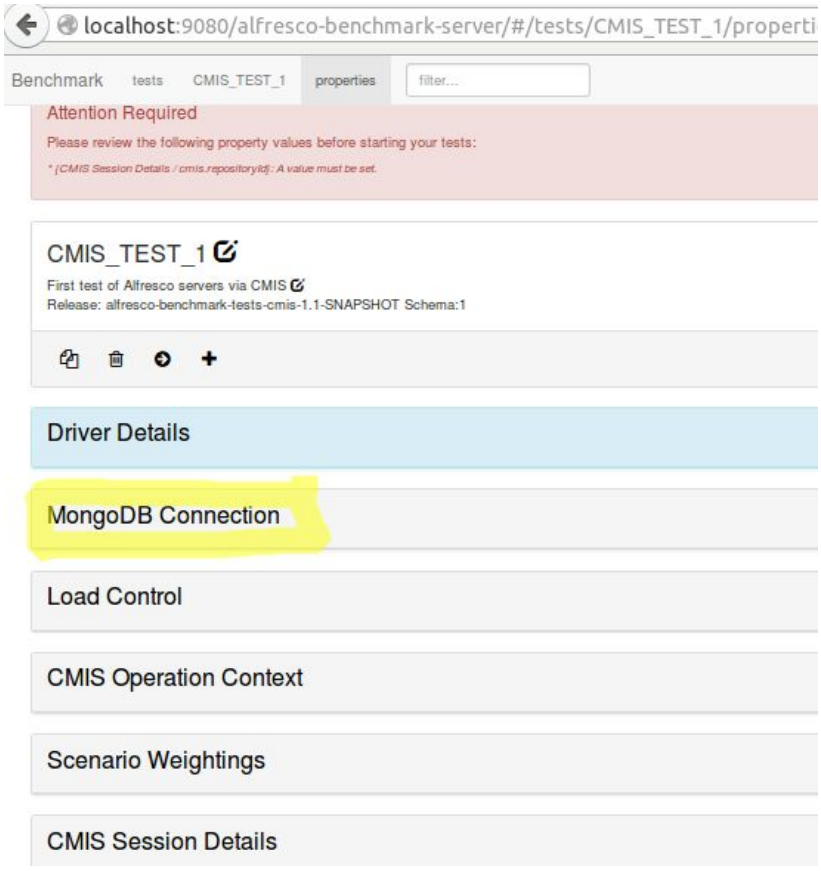
```
benchmark-cmis$ mvn tomcat7:run -Dmongo.config.host=localhost
-Dbm.tomcat.port=9091
...
INFO: Starting ProtocolHandler ["http-bio-9091"]
```

We are now ready to go back to the Management server interface and create a new CMIS test definition. In the following dialog, click the big **+**:

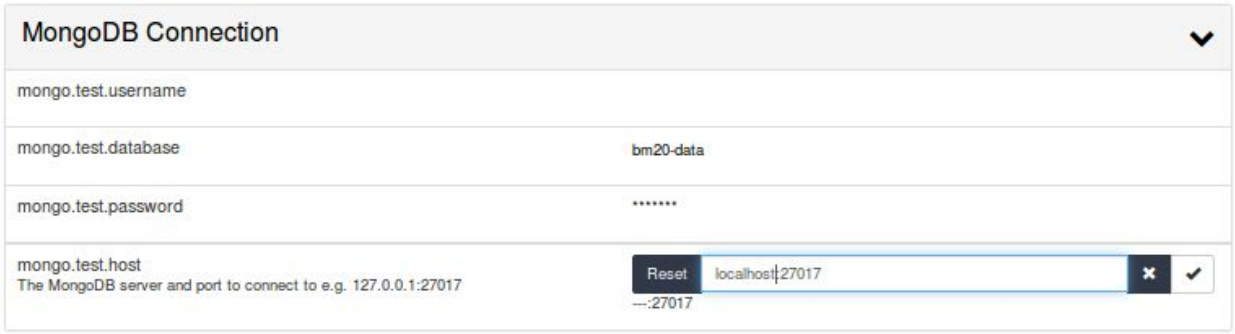


This will bring up a new dialog where we can specify a name and description for the test. After that we select which test definition (i.e. WAR) that implements it:

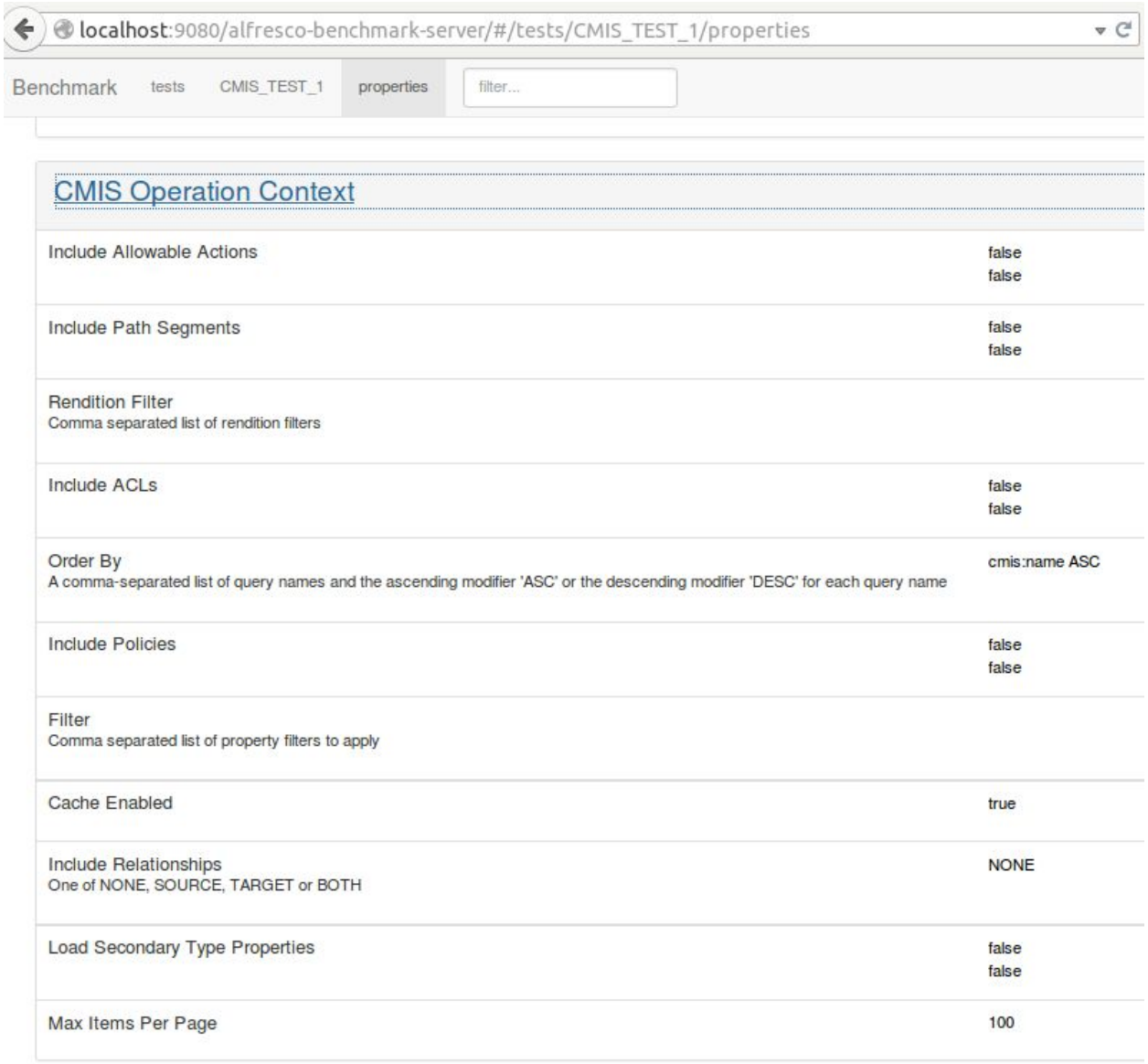
We can see that there are currently two drivers up and running that supports the CMIS test suite. Click **OK** to save this test definition, you will then see the following screen:



It is telling us that we need to also configure a location for where all the test data should end up when we run tests, it can be a different MongoDB instance if we wanted to (if you got sensitive client data for example). In our case we are going to use the same local MongoDB instance. Click the **MongoDB Connection** section to set this up:



There are a number of other properties sections where we can configure stuff, most are standard and look the same for all load tests implementations, but there are also sections specific to the load test implementation. One of the properties sections that is unique for the CMIS test is called **CMIS Operation Context**:



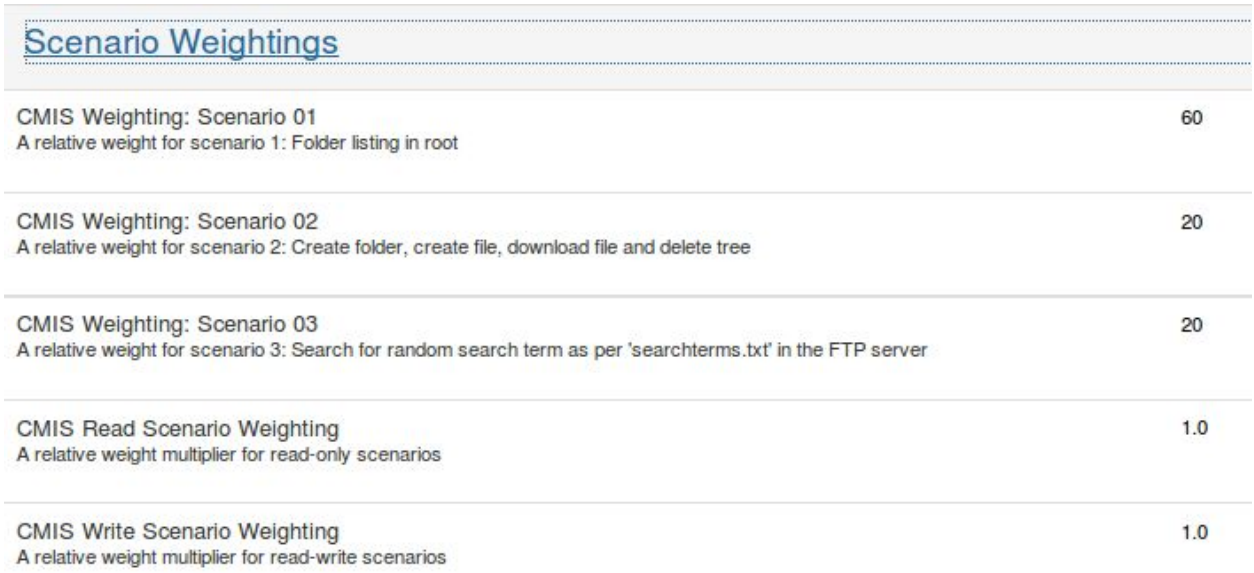
If you are a bit familiar with the CMIS standard you will recognize these properties. A lot of them control what will be returned by the CMIS Server (i.e. Alfresco) as a response to a request. All these payload config properties are set to `false` by default, so minimal data is going to be returned with each CMIS request.

If you for example want relationship/association information returned with object information then set the "`Include Relationships`" property to `true`. Fiddling around with these parameters will impact performance of the test. So make sure you figure out what your solution will need, and then configure appropriately.

Note also that the "`Cache Enabled`" property is set to `true`, so you will not always hit the repository when making a CMIS request, some request will return cached responses. This is

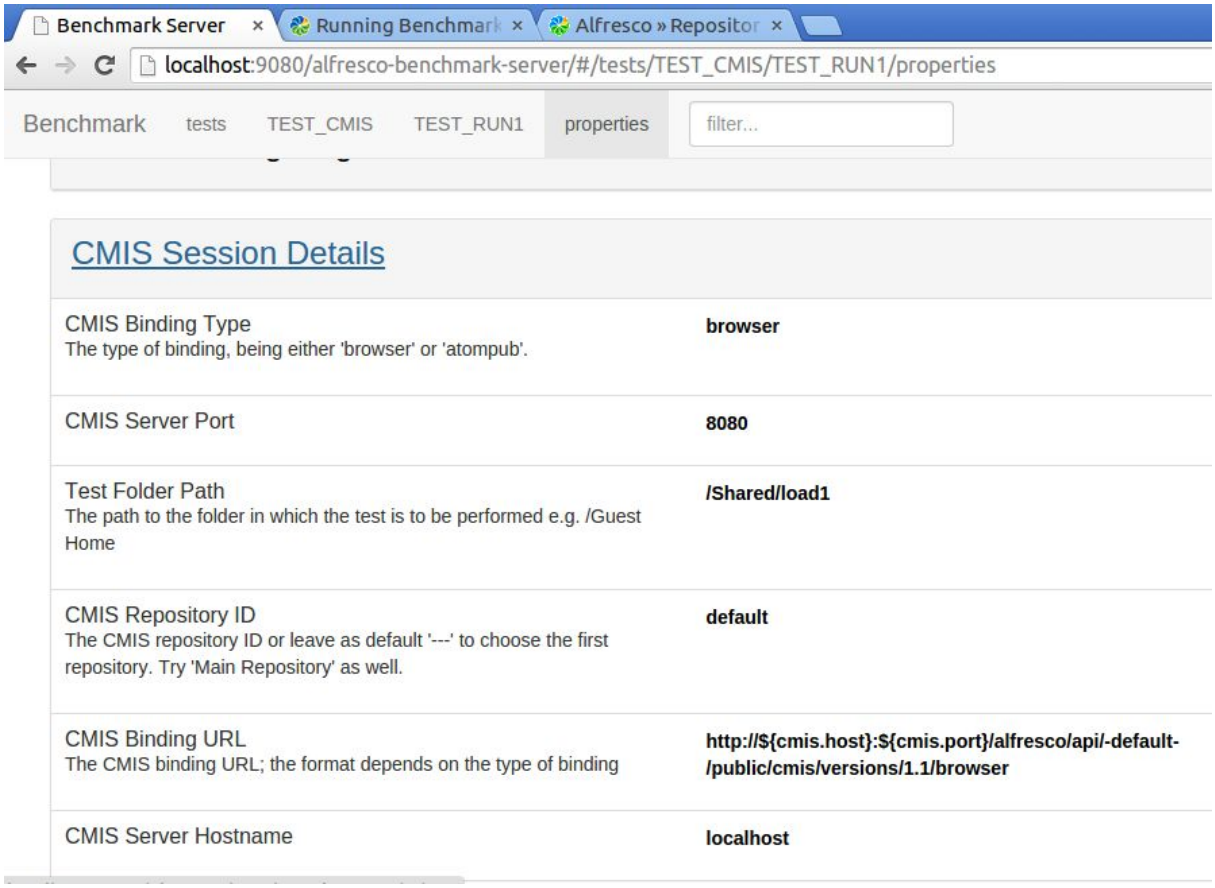
important to know as if your Alfresco solution will use CMIS, and not have cache enabled, then you need to make sure it is disabled here to so you get proper benchmarking.

We also have test **Scenario Weightings** properties:



The relative weight indicates how likely it is that a particular scenario is selected for execution relative to other scenarios. For example, if `Scenario 01` has a weight of 60, it is three times more likely to be randomly selected as, say, `Scenario 03` with a weight of 20.
We will run with the default weightings.

The next group of properties that are specific to the CMIS test are called **CMIS Session Details**:



We have done a few changes here compared to how it looks like by default. The default configuration actually points to Alfresco's public CMIS test server, we need to have it point to our local server.

These properties determine how the CMIS test clients (i.e. the Drivers) will connect to the Alfresco server and how they will communicate with it via the CMIS protocol. There are 3 so called bindings that can be used, **atompub** (XML), **browser** (JSON), and **webservice** (SOAP). The most bandwidth efficient binding is probably the `browser` one as the data packages sent with requests and responses are much more lightweight than for the XML based bindings.

We set the `CMIS Server Port` to `8080` as that is the default port that Apache Tomcat will run on with Alfresco after a standard package installation. Also, the `CMIS Server Hostname` is set to `localhost`. These properties are used when the CMIS Binding URL is put together. Be sure to connect to the CMIS endpoint using a tool like the CMIS Workbench to validate that the URL is present and not hidden by firewall rules, etc.

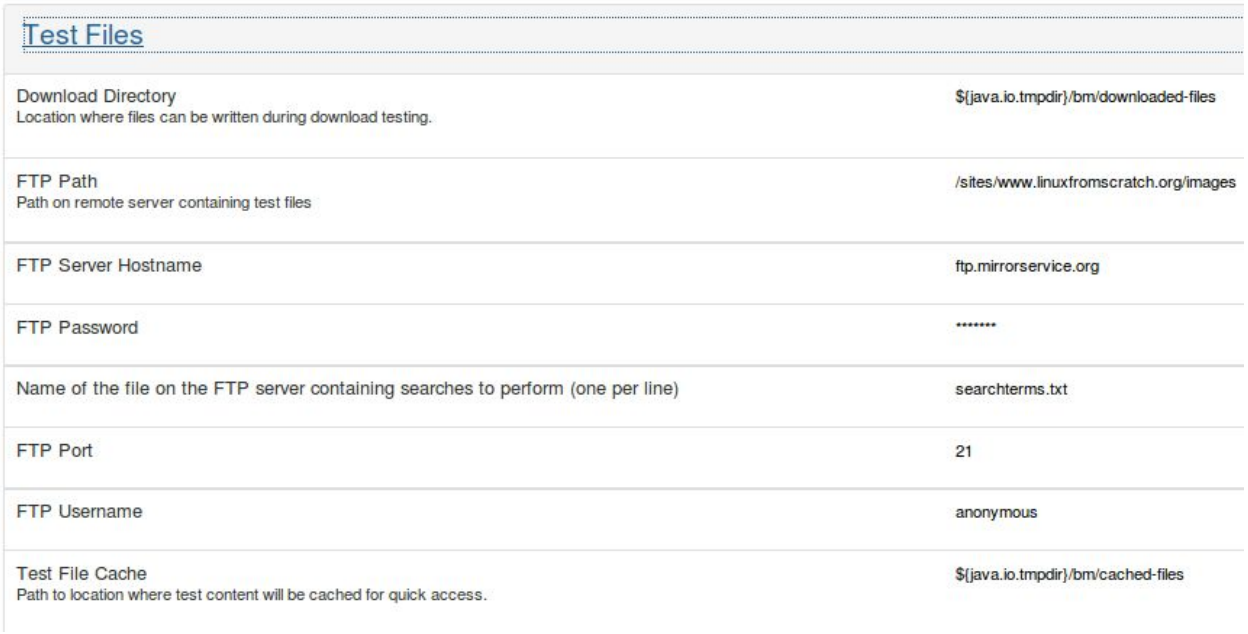
We are also setting a new `Test Folder Path` to a folder where the CMIS tests can mess around and create folders, upload files etc. You need to create this test path via the Share Web

application (i.e. http://localhost:8080/share), create a folder called `load1` in the **Shared Files** section.

Alfresco's `CMIS Repository Identifier` is called `default` so we set that too.

Then we update the `CMIS Binding URL` to http://${cmis.host}:${cmis.port}/alfresco/api/-default-/public/cmis/versions/1.1/browser, which is the correct URL when using the browser binding with an Alfresco server (note. this URL differ between CMIS servers, so if you would like to use the CMIS Test suite with another vendor, then find out the relevant browser binding URL. You would also need to populate the server with users, as the Sign-Up test suite is specific to Alfresco).

The CMIS test will include events that upload files to the Alfresco repository. You might be wondering from where these files are coming? This is configured via the **Test Files** properties group:

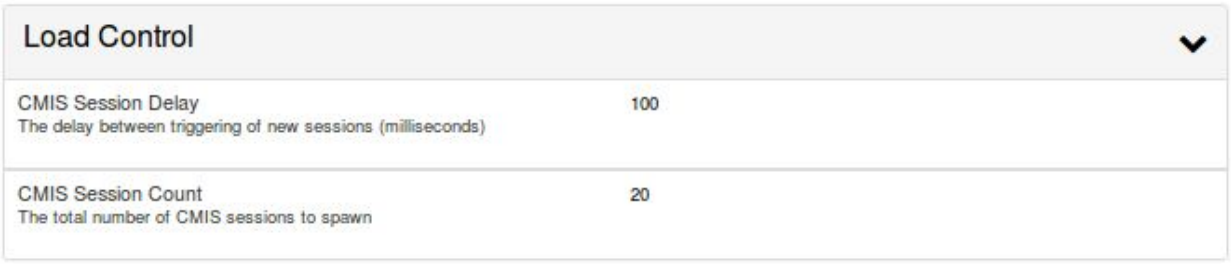


We don't need to change anything here, we just use the default FTP download site. The FTP site is used by the Drivers for loading files, downloads names from the FTP, and talking to the database about what files it has for indexing. You can use your own FTP and fill it up with your specific content files for testing your specific solution.

Another thing that you will be asking yourself is, how can we control how many concurrent users are hitting the Alfresco system, we are likely to have requirements along the lines "The system should support 25 concurrent users". It might be tempting to start fiddling around with the **Events and Threads** property group:

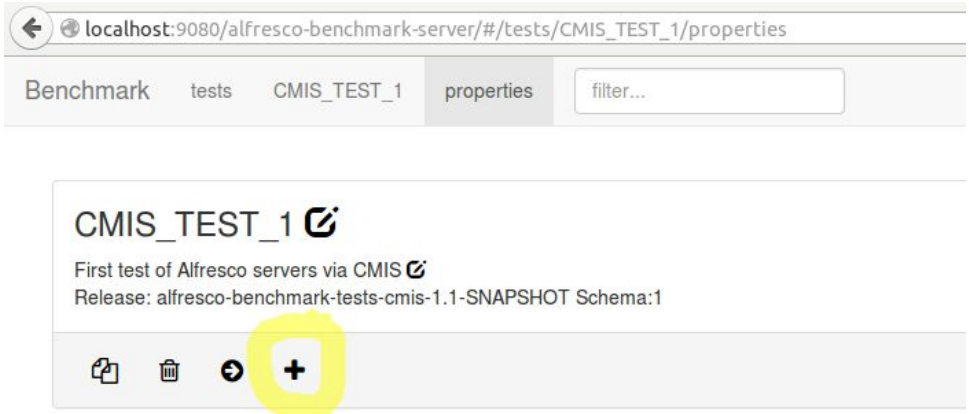
But these properties are used by the Benchmark Drivers when they process events and are sufficient for most hardware that you might be running the Driver servers on. If you have an extremely powerful Driver machine, then you can start looking at re-configuring these properties. They do not determine the rate of how a test is run, how each test is run is specific to the test.

So if you want to simulate multiple users hitting the Alfresco servers with some specific test, then you have to build this into the test, maybe offering a new property group to control concurrent users etc. For the CMIS test suite this is handled by the **Load Control** properties group:

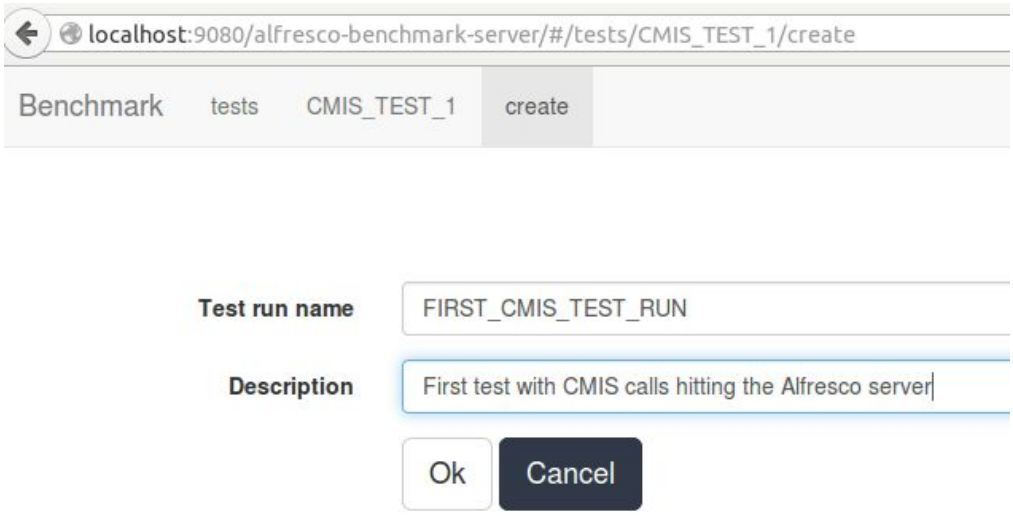


This is where we can configure how many user sessions we should create, which basically represents the number of concurrent users we want. So if we wanted to simulate 200 concurrent users we would set the `CMIS Session Count` to 200. And we can configure the delay between the start of each user session with the `CMIS Session Delay` property.

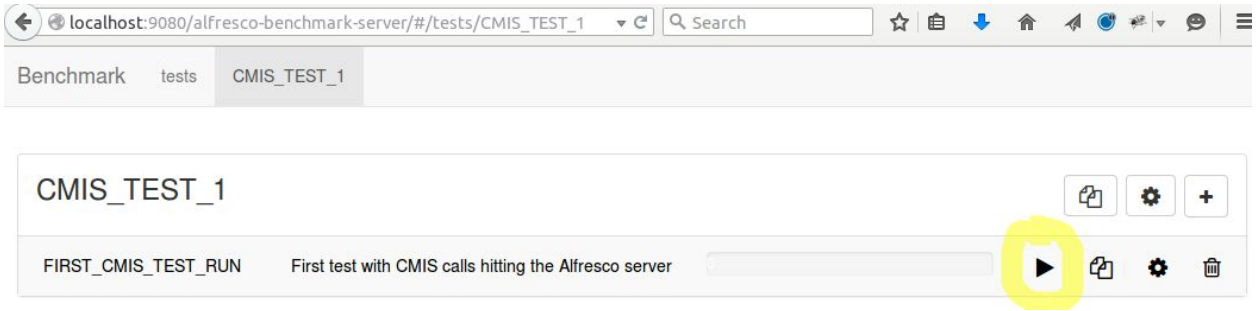
Now to run a test based on this load test configuration we need to create a test run. We do this by clicking on the **+** at the top of the screen, just under the summary and description of the test:



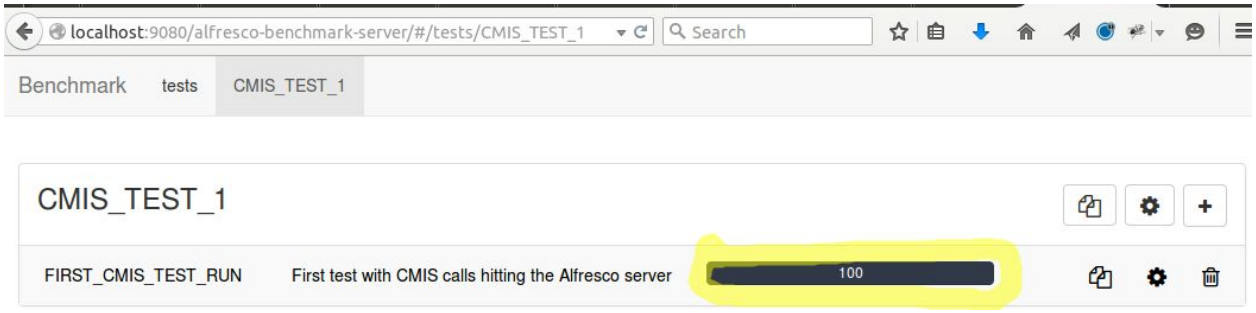
This presents the following screen where we can give the test run a name and description:



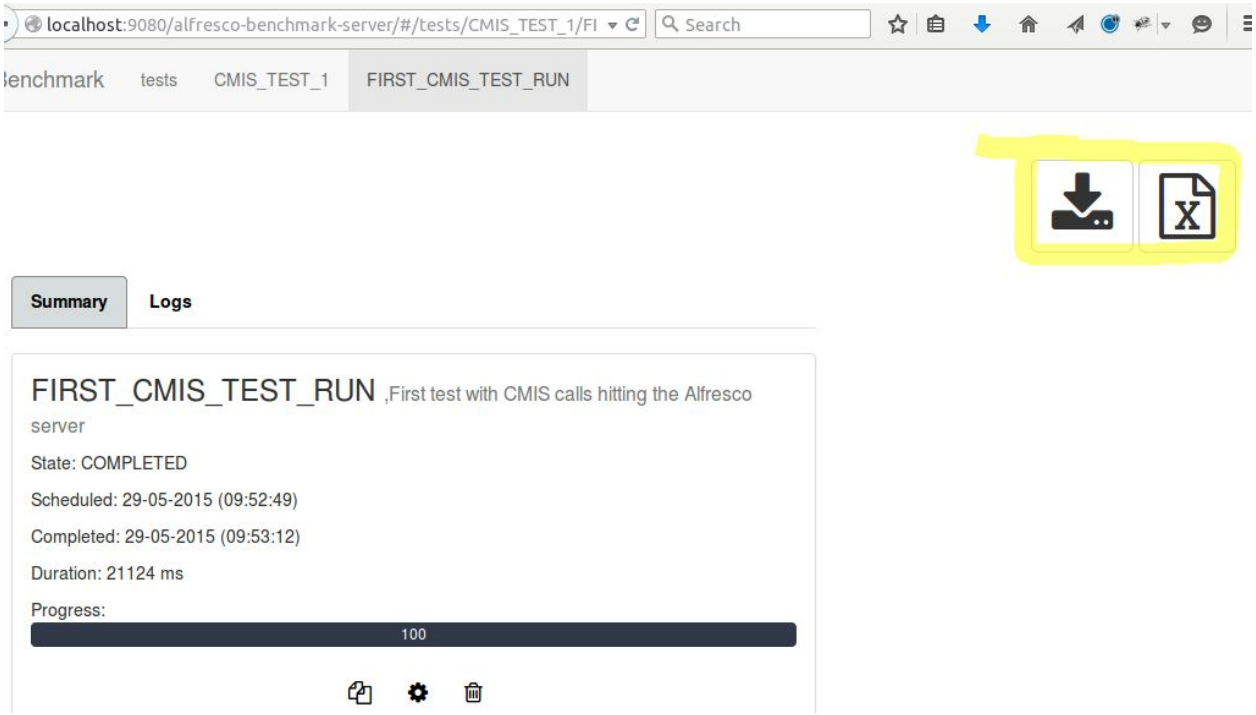
Note that the Test Definition name and the Test Run name makes up the name of the MongoDB collection where the Test Data will be stored, in this case `bm20-data/Collections/CMIS_TEST_1.FIRST_CMIS_TEST_RUN.*`. Click **OK** and you will see the screen where the test runs can be started:



Click the arrow to start the load test, it will be executed by both of the Benchmark Drivers that we started. When the test is finished you will see the following screen:



Here you can click on the test row to navigate to a screen where you can download the load test result as a CSV or Excel file:



This completes the whole process of running the CMIS load test from the Benchmark Management server.

Running the Data Load test suite

Creating Alfresco Share sites and uploading content to their Document Libraries are very common tasks in an Alfresco solution. This group of tests can be used to create loads of sites with folders and files in their Document Libraries. This test depends on the Sign-Up test that we have previously walked through. If you have not run the Sign-Up test do so now so you have some test users.

The Data Load test is available in Subversion, use the following command to download the source code:

```
$ svn co https://svn.alfresco.com/repos/alfresco-open-mirror/benchmark/tests/dataload/
alfresco-benchmark-dataload
$ cd alfresco-benchmark-dataload
alfresco-benchmark-dataload$ cd trunk/
alfresco-benchmark-dataload/trunk$
```

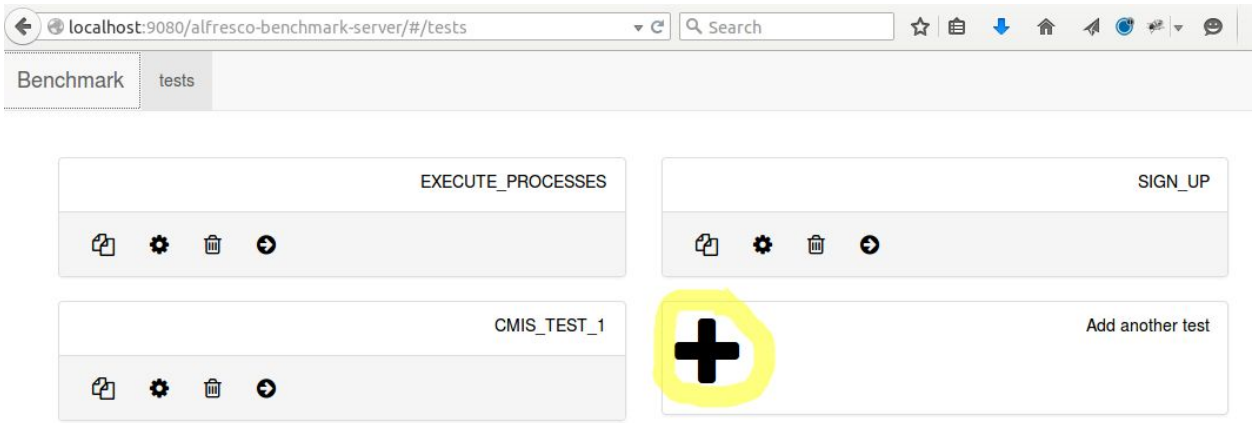
We will kick off 2 drivers with this test. Start Driver 1 as follows:

```
alfresco-benchmark-dataload/trunk$ mvn tomcat7:run
-Dmongo.config.host=localhost
…
INFO: Starting ProtocolHandler ["http-bio-9086"]
```

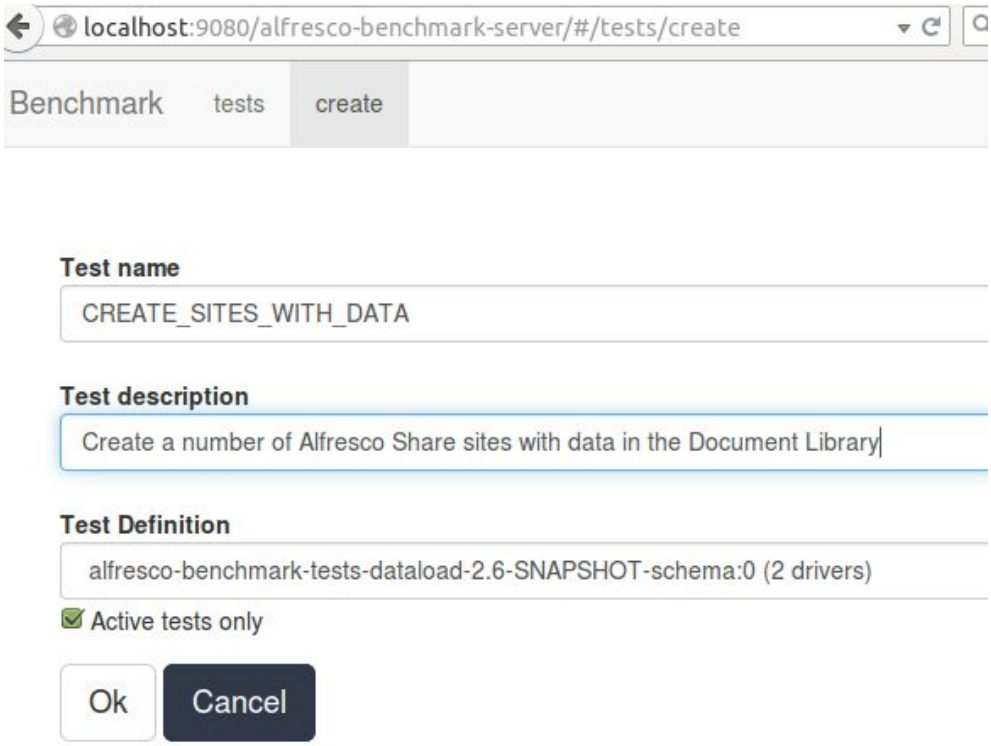
To run the Driver server from maven we again use the `tomcat7-maven-plugin`. For more information about the Tomcat plugin configuration see the project file. It kicks off an embedded Apache Tomcat instance with the Data Load test suite web application deployed. To kick off a second Driver just use a different console window and supply a different port number:

```
alfresco-benchmark-dataload/trunk$ mvn tomcat7:run
-Dmongo.config.host=localhost -Dbm.tomcat.port=9087
...
INFO: Starting ProtocolHandler ["http-bio-9087"]
```

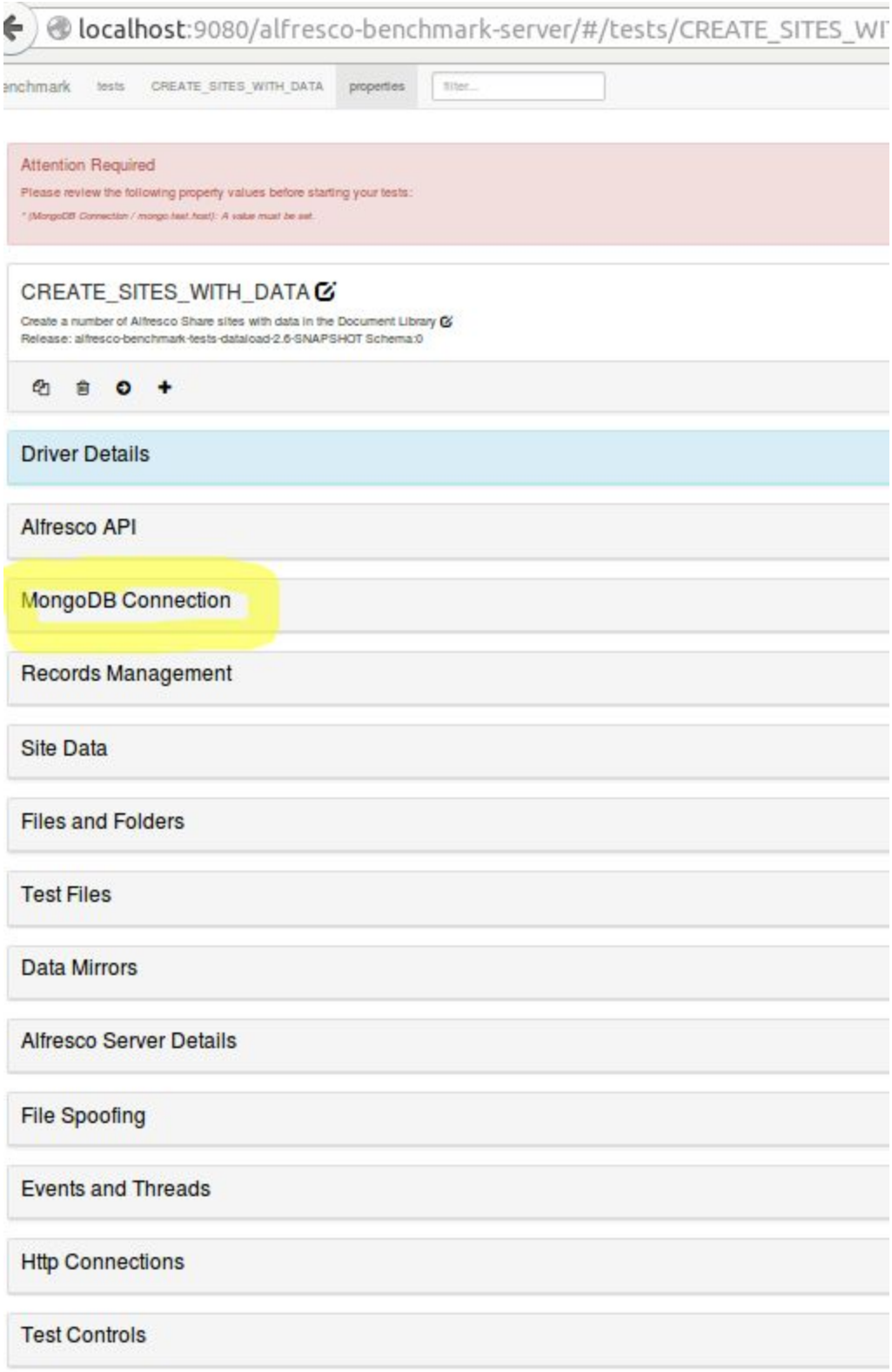
We are now ready to go back to the Management server interface and create a new CMIS test definition. In the following dialog, click the big **+**:



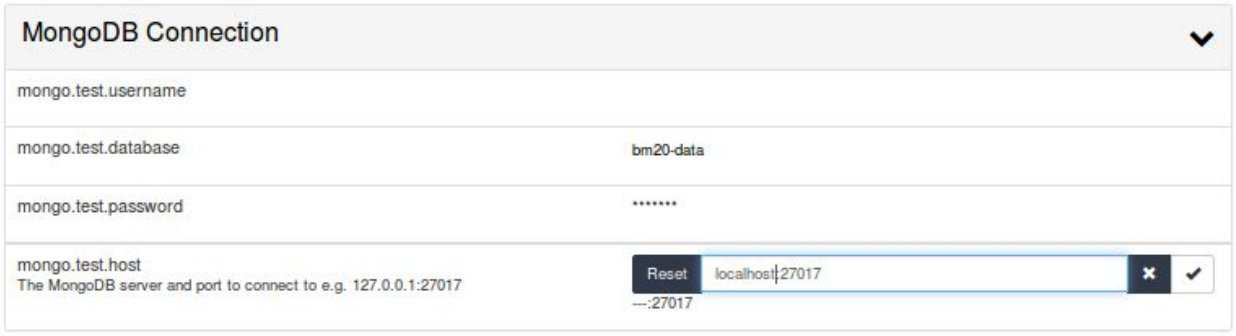
This will bring up a new dialog where we can specify a name and description for the test. After that we select which test definition (i.e. WAR) that implements it:



We can see that there are currently two drivers up and running that supports the Data Load test suite. Click **OK** to save this load test, you will then see the following screen:


localhost:9080/alfresco-benchmark-server/#/tests/CREATE_SITES_WI
enchmark
tests
CREATE_SITES_WITH_DATA
properties
filter...
Attention Required
Please review the following property values before starting your tests:
* (MongoDB Connection / mongo.test.host): A value must be set.
CREATE_SITES_WITH_DATA
Create a number of Alfresco Share sites with data in the Document Library
Release: alfresco-benchmark-tests-dataload-2.6-SNAPSHOT Schema:0
Driver Details
Alfresco API
MongoDB Connection
Records Management
Site Data
Files and Folders
Test Files
Data Mirrors
Alfresco Server Details
File Spoofing
Events and Threads
Http Connections
Test Controls

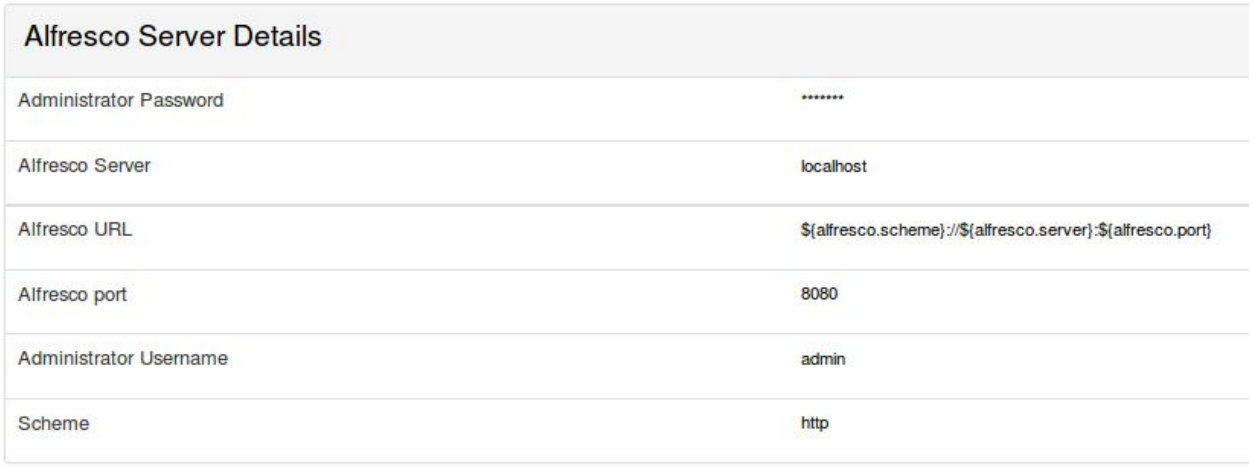
It is telling us that we need to also configure a location for where all the test data should end up when we run tests, it can be a different MongoDB instance if we wanted to. In our case we are going to use the same local MongoDB instance. Click the **MongoDB Connection** section to set this up:



There are a number of other properties sections where we can configure stuff, most are standard and look the same for all load tests implementations. For example, the **Alfresco API** properties section is another group of properties that can be re-used by different test implementation:

The Data Load test uses the CMIS protocol to talk to Alfresco when it creates folders and files in the Document Library of a Site. So we configure the CMIS URL to use for that. When the Data Load test creates the Share Sites it cannot use CMIS, as the Site concept is not part of the CMIS standard, so it uses proprietary Alfresco APIs to do that. This is where the `HTTP URL Context`, `Service Servlet name`, and `API Servlet name` properties comes into play. You would normally not have to change any of these properties.

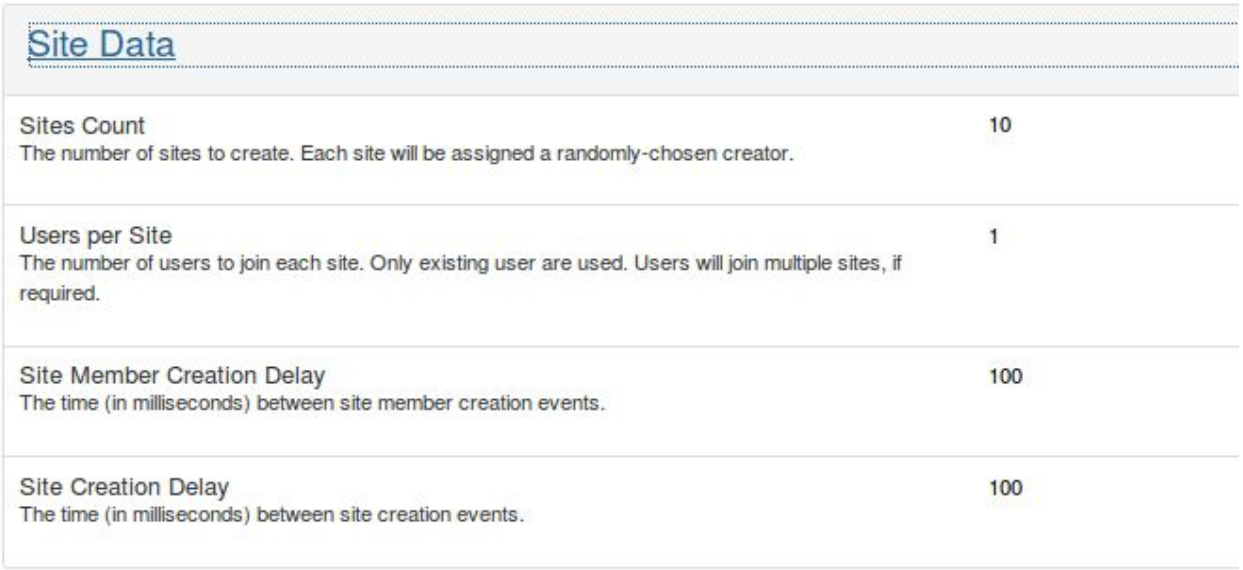
Related to the Alfresco API properties are the **Alfresco Server Details**:



This is where you set the connection and authentication information to use when communicating with the Alfresco server, whether via CMIS or proprietary protocol. The username/password is set to admin/admin, change it if you specified something else. If you want to test over a secure connection then change `Scheme` to `https` and `Alfresco port` to `8443`.

Note that it is important that the `Alfresco Server` hostname that is specified is the same hostname used by the Sign-Up test, so that the user data mirror name matches (which is `mirrors.localhost.users` from our previous Sign-Up test run). As mentioned before, this value can be set once at the test definition level when performing repeated runs against the same server. The hostname must be visible to the Driver server instances where the tests are deployed.

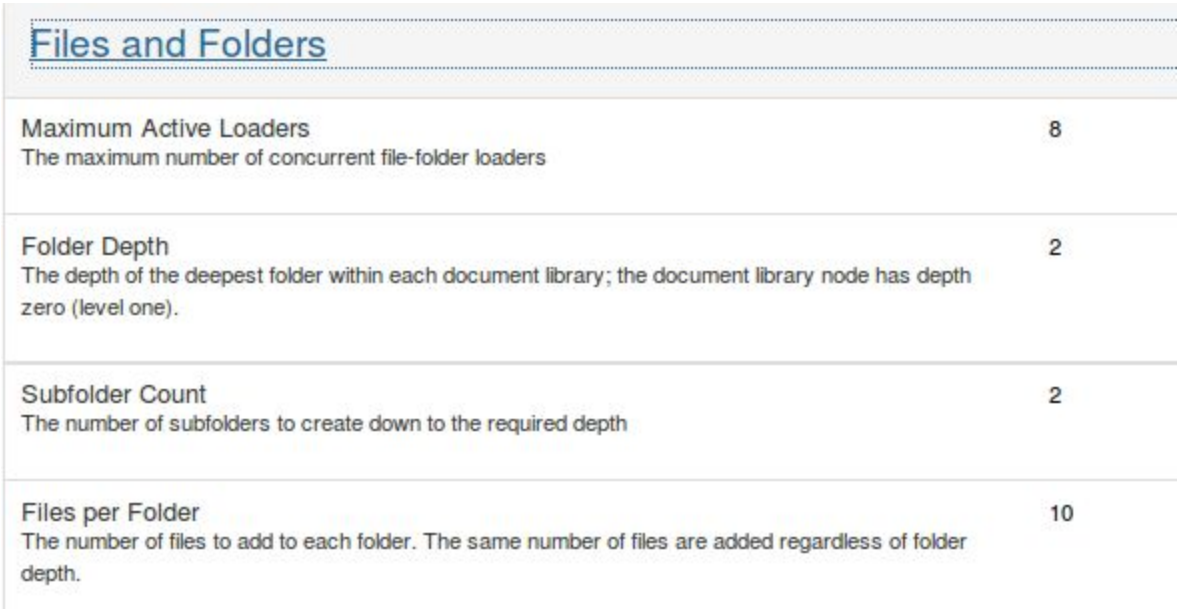
The **Site Data** properties group can be used to control the number of Alfresco Share sites that are created, this property group is specific to the Data Load test:



You can tweak these settings to model the number of sites and members that you will have in your Alfresco solution.

If you are using the Records Management (RM) module with Alfresco you can test the RM Site with the Data Load test. Specific configuration of the RM Site, such as how to create files randomly in the File-plan, can be done via the **Records Management** properties group. In this test run we are not going to do any RM testing.

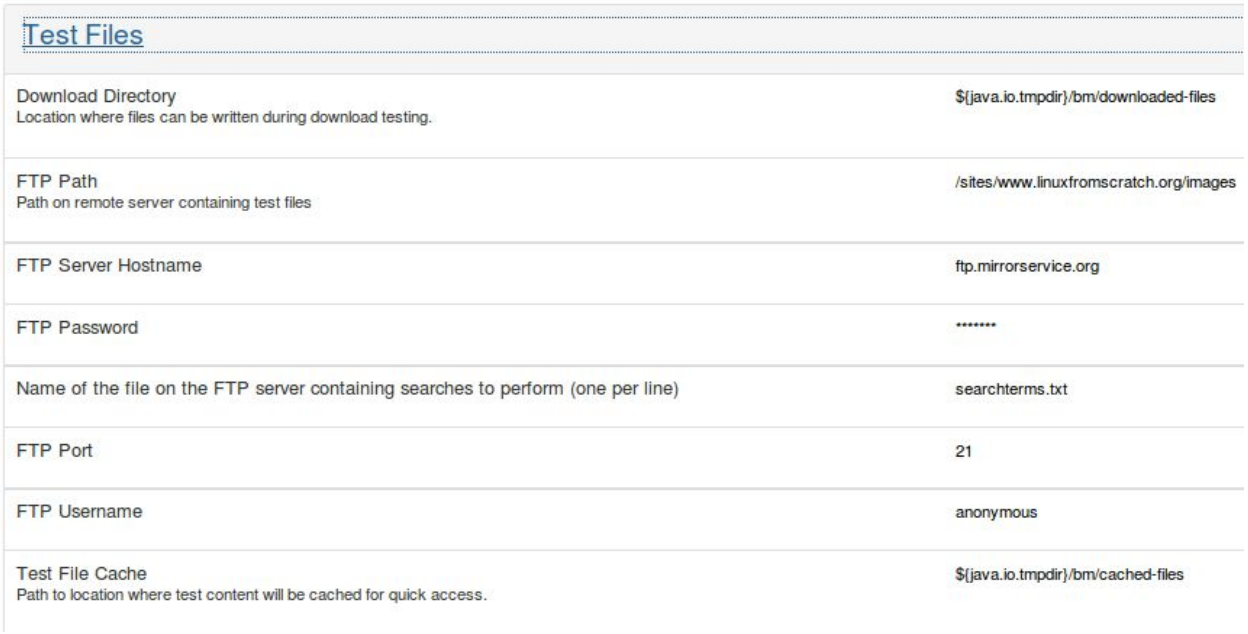
The number of folders and files that are created in each site is controlled from another properties group called **Files and Folders**:



You can control how many folders are created and to what depth, and how many files should be created in each folder. Use the `Maximum Active Loaders` property to simulate concurrent users creating folders and files. For example, if you expect 200 users to concurrently create folders and files, then set this property to 200.

If you don't want to create any folders in the Document Library of the site set the `Folder Depth` to `0`. If you set the folder depth to greater than 0 then the sub-folders are created using a random site member/user in a single CMIS session. The number of sub-folders in each folder is controlled by the `Subfolder Count`. Each folder is then loaded with files as per the `Files per Folder` setting. The file upload is a remote upload of a real document using CMIS and a random site user/member within a single CMIS session.

The files to be uploaded in each folder is coming from an FTP site configured via the **Test Files** properties group:



Test Files

Download Directory Location where files can be written during download testing.	${java.io.tmpdir}/bm/downloaded-files
FTP Path Path on remote server containing test files	/sites/www.linuxfromscratch.org/images
FTP Server Hostname	ftp.mirrorservice.org
FTP Password	*******
Name of the file on the FTP server containing searches to perform (one per line)	searchterms.txt
FTP Port	21
FTP Username	anonymous
Test File Cache Path to location where test content will be cached for quick access.	${java.io.tmpdir}/bm/cached-files

We don't need to change anything here, we just use the default FTP download site. The FTP site is used by the Drivers for loading files, downloads names from the FTP, and talking to the database about what files it has for indexing. You can use your own FTP and fill it up with your specific content files for testing your specific solution.

By now you might be thinking along the lines of - what if I got tests that should include thousands of sites and each site's document library should contain tens of thousands of files. Isn't that going to take a long time to actually run? For example, if we want to create 10000 sites with 50000 files in each, that's going to be 500 million files. And let's say you can load 100 files / second (which is quite optimistic with a remote API...). That's going to be upwards of 60 days to load…, not very practical.

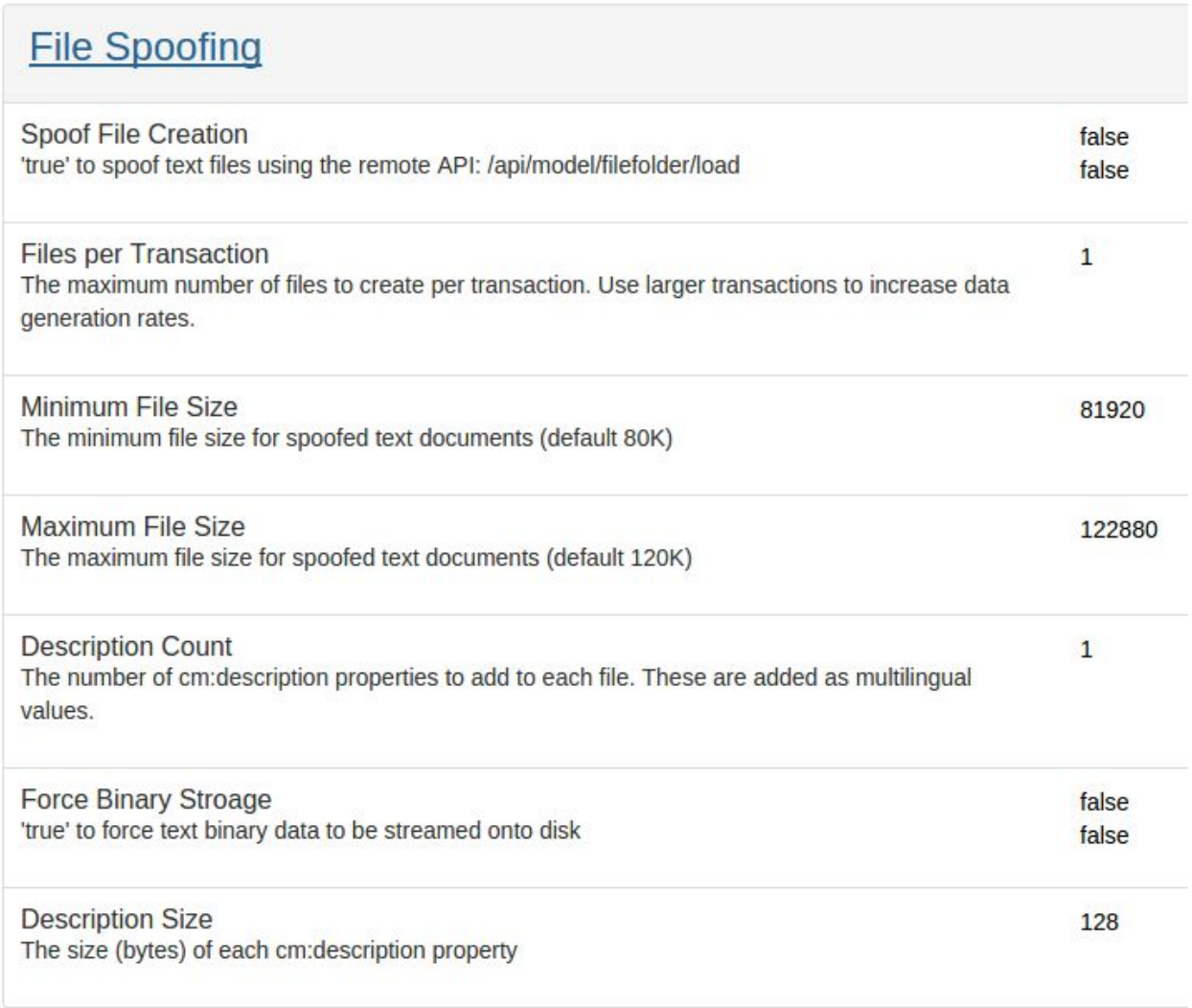
There is help on the way. In Alfresco 5.1 there is support for something called file spoofing. What that means is that you can tell Alfresco to create loads of files for you without actual content bytes. This is handled via a new class called `FileFolderLoader` and the method called `createFiles`, which has the following signature:

```
public int createFiles(
        final String folderPath,
        final int fileCount,
        final int filesPerTxn,
        final long minFileSize, long maxFileSize,
        final long maxUniqueDocuments,
        final boolean forceBinaryStorage,
        final int descriptionCount, final long descriptionSize
```

So you can give it the folder you want to load with files, how many files should be generated, transaction boundaries, file sizes (if you decide to have content bytes), description etc. This is

obviously going to be very fast if you skip content bytes for the files. You might have use cases where you just want to search file metadata and you need loads of files. This is also going to be much faster when you are using content bytes (i.e. `forceBindaryStorage` = true). With this method you bypass all remote APIs and downloads.

When you use the Data Load test suite there is a group of properties called **File Spoofing** that can be used to activate the `FileFolderLoader` component and in-place file creation:



File Spoofing	
Spoof File Creation 'true' to spoof text files using the remote API: /api/model/filefolder/load	false false
Files per Transaction The maximum number of files to create per transaction. Use larger transactions to increase data generation rates.	1
Minimum File Size The minimum file size for spoofed text documents (default 80K)	81920
Maximum File Size The maximum file size for spoofed text documents (default 120K)	122880
Description Count The number of cm:description properties to add to each file. These are added as multilingual values.	1
Force Binary Stroage 'true' to force text binary data to be streamed onto disk	false false
Description Size The size (bytes) of each cm:description property	128

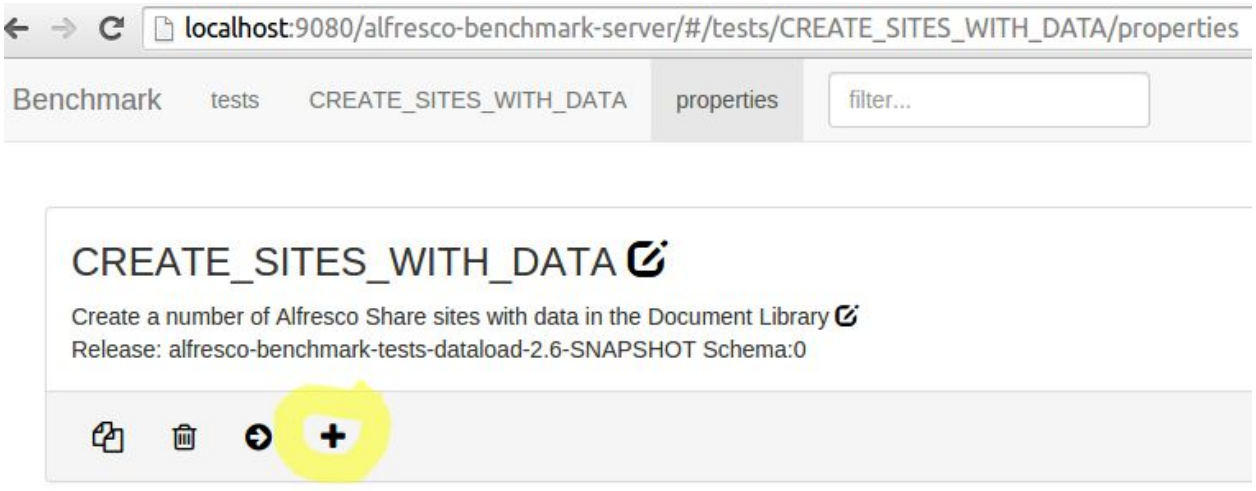
File Spoofing is turned off by default. Turn it on by setting the `Spoof File Creation` property to `true`. This will make every call to `/api/model/filefolder/load` “spoofed”. You can use the rest of the properties to control the parameters to the `FileFolderLoader.createFiles` method. We will not use file spoofing in this test run so I’m leaving these properties as is.

As with all the other test suites we have tried out, the **Events and Threads** property group cannot be used to simulate concurrent users creating sites:

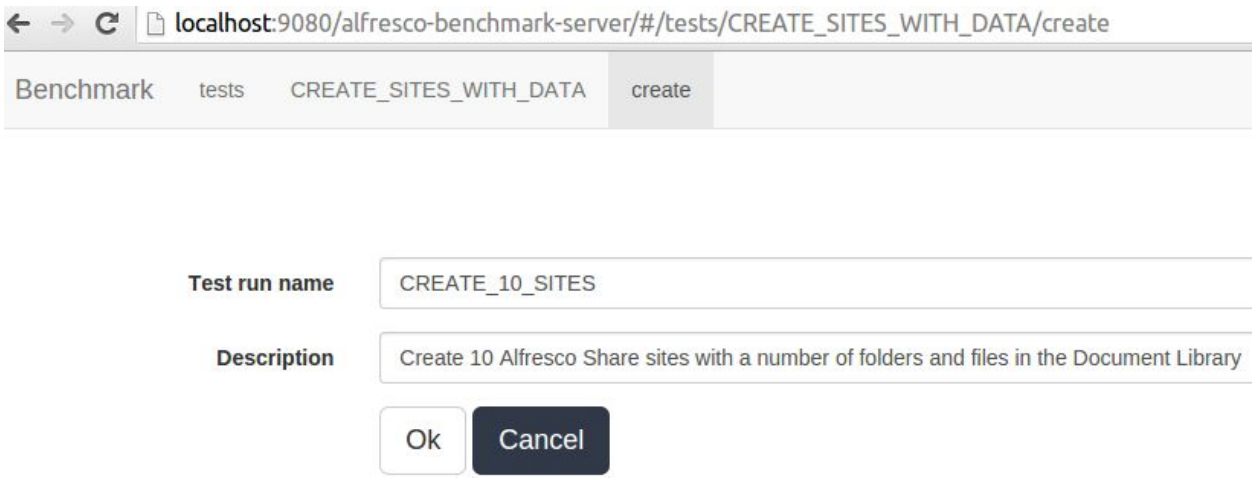
These properties are used by the Benchmark Drivers when they process events and are sufficient for most hardware that you might be running the Driver servers on. If you have an extremely powerful Driver machine, then you can start looking at re-configuring these properties. They do not determine the rate of how a test is run, how each test is run is specific to the test.

The Data Load test suite has not been designed to test creating multiple sites concurrently by different users. It is a test suite similar to Sign-Up, meaning it is used to load content/data into the Alfresco repository in preparation for running other tests, such as the Share UI test that we are going to look at when we are finished with the Data Load test run. You can however control a bit of concurrency with settings in the Files and Folders properties group we looked at earlier.

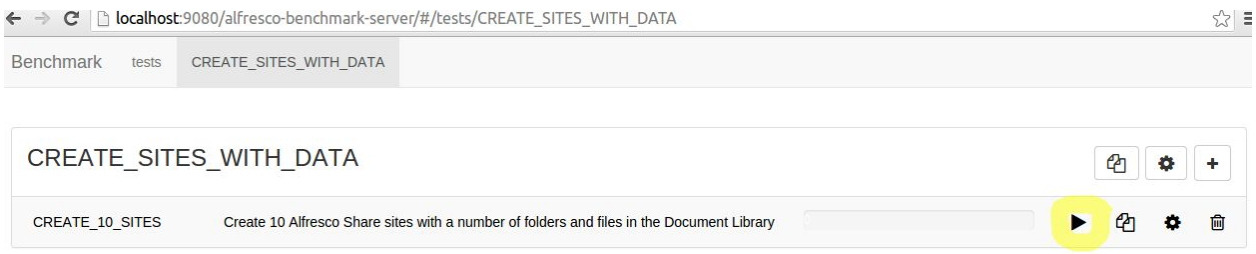
Now to run a test based on this load test configuration we need to create a test run. We do this by clicking on the **+** at the top of the screen, just under the summary and description of the test:



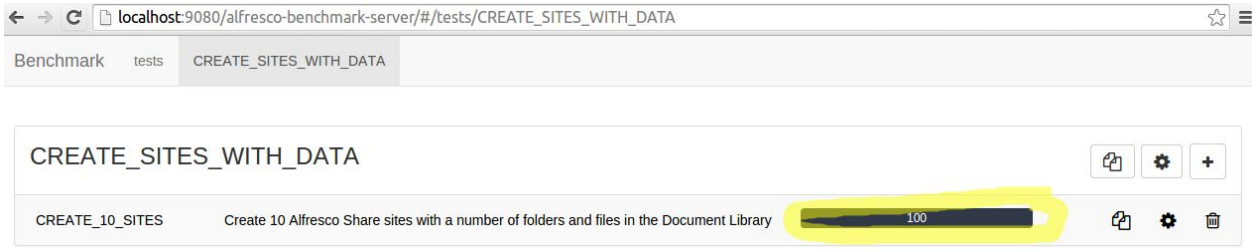
This presents the following screen where we can give the test run a name and description:



Note that the Test Definition name and the Test Run name makes up the name of the MongoDB collection where the Test Data will be stored, in this case `bm20-data/Collections/CREATE_SITES_WITH_DATA.CREATE_10_SITES.*`. Click **OK** and you will see the screen where the test runs can be started:

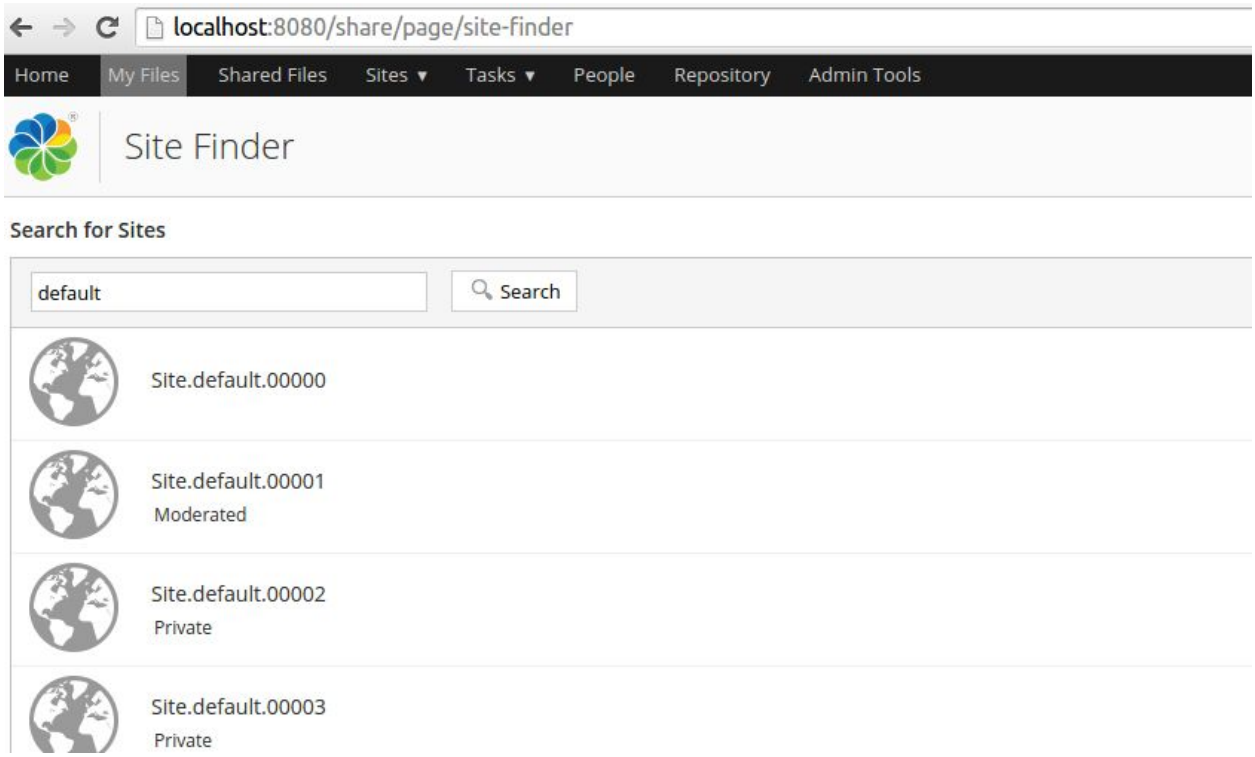


Click the arrow to start the load test, it will be executed by both of the Benchmark Drivers that we started. When the test is finished you will see the following screen:



Here you can click on the test row to navigate to a screen where you can download the load test result as a CSV or Excel file:

This completes the whole process of running the Data Load test from the Benchmark Management server. To make sure it actually work we can have a look in Alfresco Share and search for the sites, select **Sites | Site Finder**:

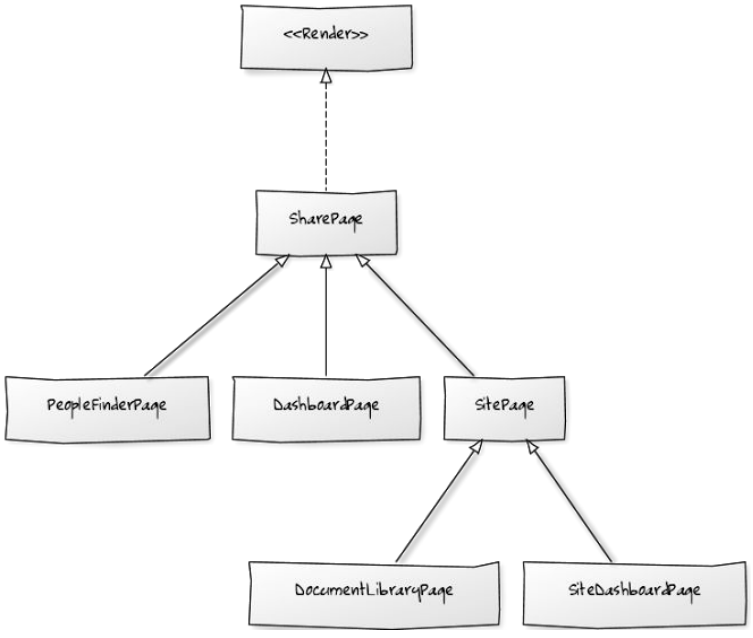


Note that sites are created with all the different visibility configurations, public, moderated, and private. The information about these sites are mirrored in the MongoDB database. If you run this test again it will not do anything as the sites are already there. Information about the mirror configuration can be found in the **Data Mirrors** property group:

Data Mirrors	
Test Files Cache Mirror Name The name of a MongoDB collection to store locally cached test content. The format is 'mirror.abc.files'.	mirrors.cached.files
Folder and Files Collection Name	mirrors.${alfresco.server}.filefolders
Sites Collection Name	mirrors.${alfresco.server}.sites
User Collection Name	mirrors.${alfresco.server}.users
Site Members Collection Name	mirrors.${alfresco.server}.siteMembers

Running the Share test suite

We have now gone through quite a few test suites that comes with the Benchmark Toolkit. What we are missing is a test suite that can be used to load test the Alfresco Share user interface. This is what the Share test suite is for. These tests are based on the the *Page Object* pattern that maps Share pages to series of objects:



"A page object wraps an HTML page, or fragment, with an application-specific API, allowing you to manipulate page elements without digging around in the HTML"
Martin Fowler

The Share test depends on the Sign-Up test that we have previously walked through. If you have not run the Sign-Up test do so now so you have some test users.

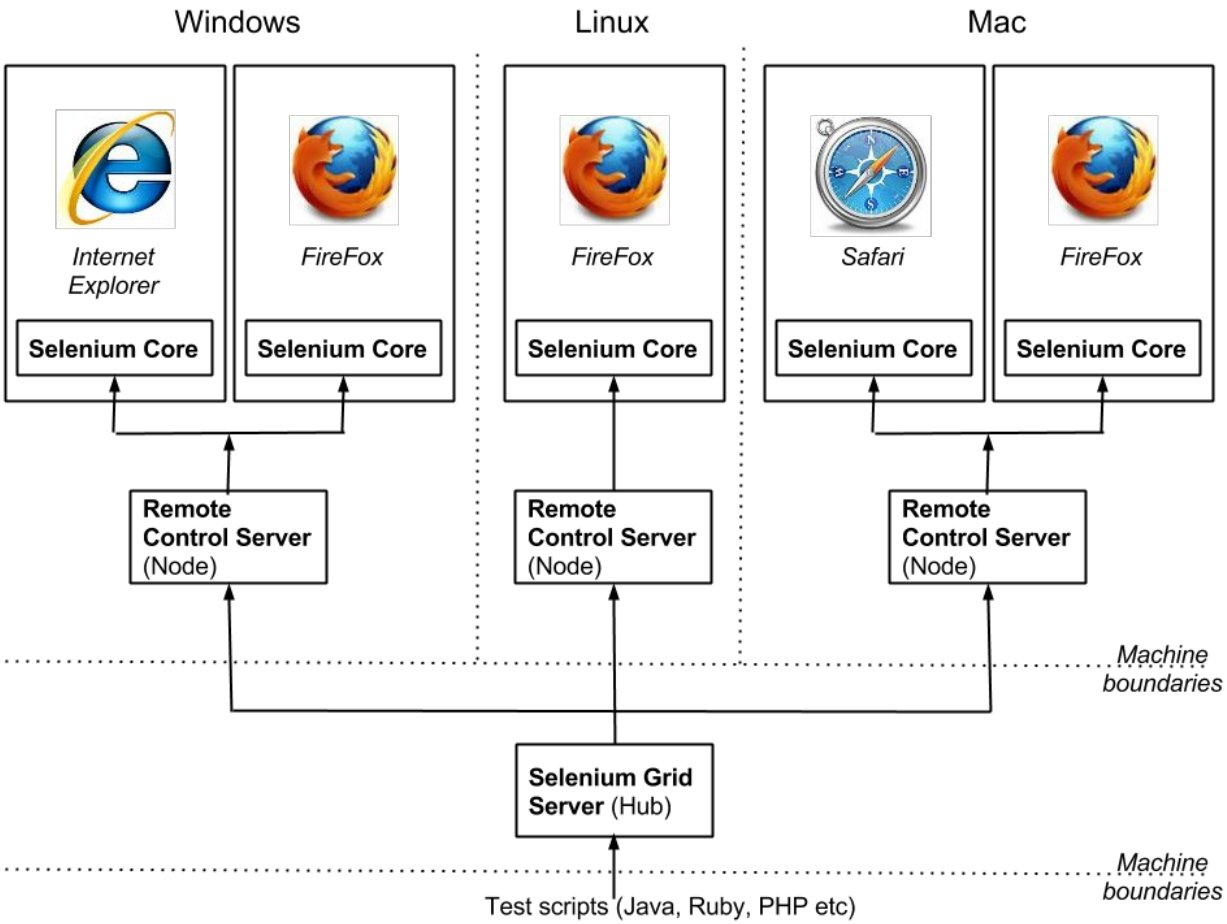
When we test the user interface we want to make sure it works with different browsers and operating systems. So we cannot just

assume that the Driver servers will be sufficient for this. We need something else that the Driver servers can work with (i.e. activate UI tests remotely). There are services out there that specialize in automated user interface testing, such as Sauce Labs or Browser Stack, but if you cannot use these commercial solutions, then what do you do?

The Share test suite uses the Selenium browser automation tools. Specifically it uses the Selenium WebDriver to create robust, browser-based regression automation test suites that can scale and distribute scripts across many environments. The Selenium WebDriver is available as a standalone server that can be configured to setup a Selenium Grid.

Selenium Grid is an open source framework for scaling your test suites across multiple browsers and browser versions, and running them in parallel. This allows not only for more complete test coverage, but also faster test cycles. Instead of running tests serially, as you might do using Selenium and whatever browser is installed on your own computer, you can spread those out across different machines to run at once, substantially reducing your overall test time.

A typical setup a Selenium Grid looks something like this:



This setup requires a lot of hardware and we are not going to set up all that here now. We will just use the middle Linux setup on the same box as we are running the other stuff (i.e. the Benchmark Management server, Driver server, and Alfresco Server). This means that we need to download the Selenium Standalone Server and then use it to start up a Selenium Grid hub and node.

Go to the Download page and grab latest JAR via link under **Selenium Standalone Server** section. Then create a directory and put the JAR in it:

```
martin@gravitonian:~/apps$ mkdir selenium-server
martin@gravitonian:~/apps$ cd selenium-server/
martin@gravitonian:~/apps/selenium-server$ cp
~/Downloads/selenium-server-standalone-2.46.0.jar .
```

Now start the hub server using the JAR:

martin@gravitonian:~/apps/selenium-server$ java -Xmx1024M -jar selenium-server-standalone-2.46.0.jar -role hub -DPOOL_MAX=50 -newSessionWaitTimeout 5000 -timeout 300
11:25:33.690 INFO - Launching Selenium Grid hub
2015-06-16 11:25:34.414:INFO:osjs.Server:jetty-7.x.y-SNAPSHOT
2015-06-16 11:25:34.447:INFO:osjsh.ContextHandler:started o.s.j.s.ServletContextHandler{/,null}
2015-06-16 11:25:34.457:INFO:osjs.AbstractConnector:Started SocketConnector@0.0.0.0:4444
11:25:34.457 INFO - Nodes should register to http://10.244.51.214:4444/grid/register/
11:25:34.457 INFO - Selenium Grid hub is up and running

Then start a node using the JAR, in a new terminal:

```
martin@gravitonian:~/apps/selenium-server$ java -Xmx2048M -jar
selenium-server-standalone-2.46.0.jar -role node -hub
http://localhost:4444/grid/register -maxSession 25 -browser
browserName=firefox,platform=ANY,maxInstances=25
11:27:20.784 INFO - Launching a Selenium Grid node
11:27:20.848 INFO - Adding browserName=firefox,platform=ANY,maxInstances=25
11:27:21.302 INFO - Java: Oracle Corporation 25.45-b02
11:27:21.302 INFO - OS: Linux 3.13.0-51-generic amd64
11:27:21.314 INFO - v2.46.0, with Core v2.46.0. Built from revision 87c69e2
11:27:21.385 INFO - Driver provider
org.openqa.selenium.ie.InternetExplorerDriver registration is skipped:
registration capabilities Capabilities [{ensureCleanSession=true,
browserName=internet explorer, version=, platform=WINDOWS}] does not match the
current platform LINUX
11:27:21.385 INFO - Driver class not found: com.opera.core.systems.OperaDriver
```

```
11:27:21.386 INFO - Driver provider com.opera.core.systems.OperaDriver is not
registered
11:27:21.440 INFO - Selenium Grid node is up and ready to register to the hub
11:27:21.467 INFO - Starting auto registration thread. Will try to register
every 5000 ms.
11:27:21.467 INFO - Registering the node to the hub:
http://localhost:4444/grid/register
11:27:21.496 INFO - The node is registered to the hub and ready to use
```

Note that you must have Firefox installed as the node is started with a configuration to use the Firefox browser when running the tests. Access the Grid Console to verify that everything is OK:

So everything looks OK. Let's move on with the Share Benchmark test download and startup.

The Share test is available in Subversion, use the following command to download the source code:

```
$ svn co
https://svn.alfresco.com/repos/alfresco-open-mirror/benchmark/tests/share/
alfresco-benchmark-share
$ cd alfresco-benchmark-share/
alfresco-benchmark-share$ cd trunk/
alfresco-benchmark-share/trunk$
```

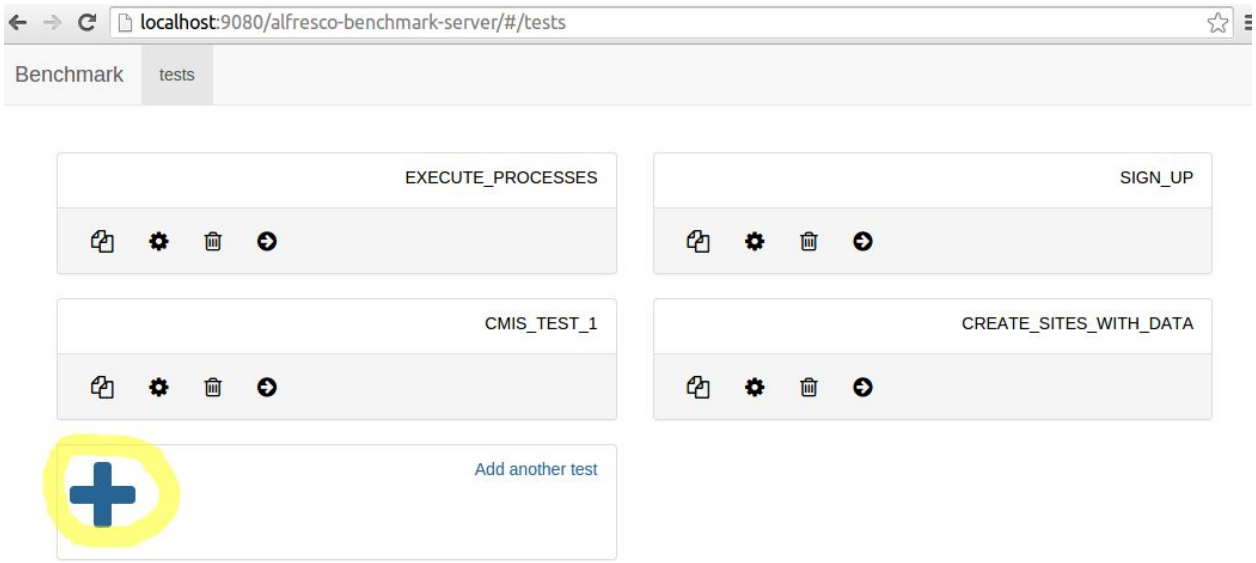
We will kick off 2 drivers with this test. Start Driver 1 as follows:

```
alfresco-benchmark-share/trunk$ mvn tomcat7:run -Dmongo.config.host=localhost
…
INFO: Starting ProtocolHandler ["http-bio-9087"]
```

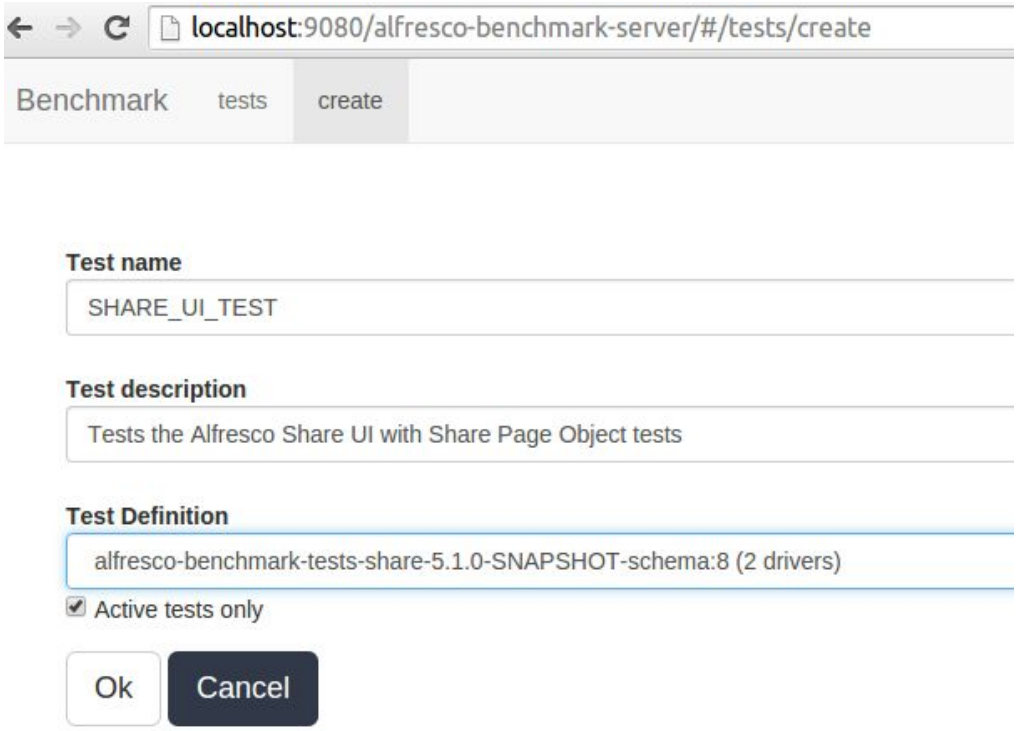
To run the Driver server from maven we again use the `tomcat7-maven-plugin`. For more information about the Tomcat plugin configuration see the project file. It kicks off an embedded Apache Tomcat instance with the Share test suite web application deployed. To kick off a second Driver just use a different console window and supply a different port number:

```
alfresco-benchmark-share/trunk$ mvn tomcat7:run -Dmongo.config.host=localhost
-Dbm.tomcat.port=9088
...
INFO: Starting ProtocolHandler ["http-bio-9088"]
```

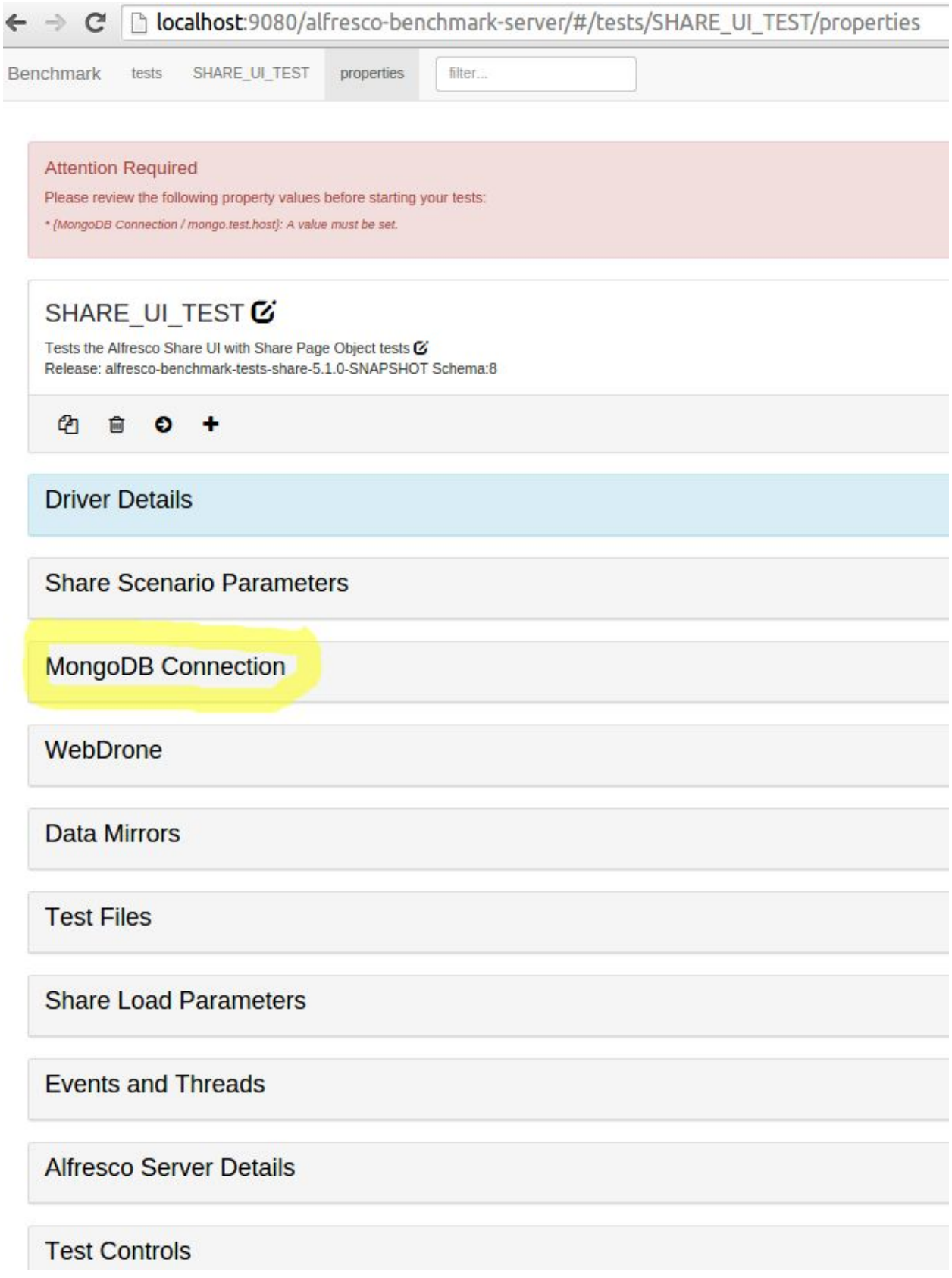
We are now ready to go back to the Management server interface and create a new Share test definition. In the following dialog, click the big **+**:



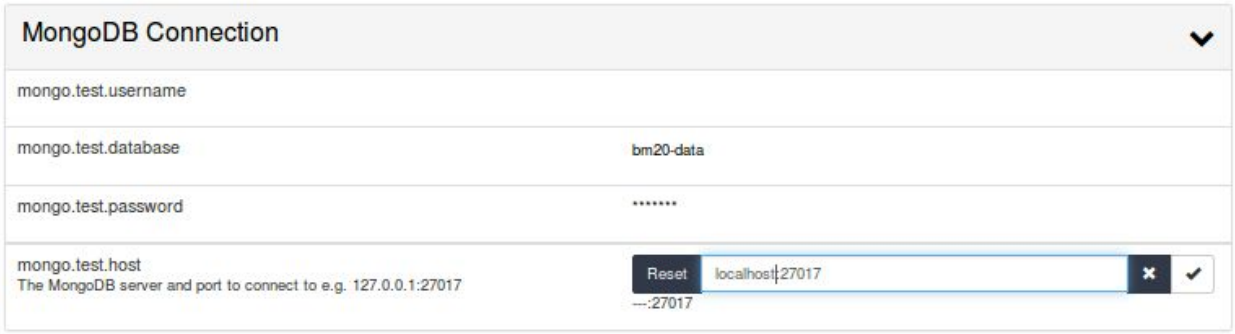
This will bring up a new dialog where we can specify a name and description for the test. After that we select which test definition (i.e. WAR) that implements it:



We can see that there are currently two drivers up and running that supports the Share test suite. Click **OK** to save this load test, you will then see the following screen:


localhost:9080/alfresco-benchmark-server/#/tests/SHARE_UI_TEST/properties
Benchmark
tests
SHARE_UI_TEST
properties
filter...
Attention Required
Please review the following property values before starting your tests:
* {MongoDB Connection / mongo.test.host}: A value must be set.
SHARE_UI_TEST
Tests the Alfresco Share UI with Share Page Object tests
Release: alfresco-benchmark-tests-share-5.1.0-SNAPSHOT Schema:8
Driver Details
Share Scenario Parameters
MongoDB Connection
WebDrone
Data Mirrors
Test Files
Share Load Parameters
Events and Threads
Alfresco Server Details
Test Controls

It is telling us that we need to also configure a location for where all the test data should end up when we run tests, it can be a different MongoDB instance if we wanted to (if we for example have sensitive client data). In our case we are going to use the same local MongoDB instance. Click the **MongoDB Connection** section to set this up:



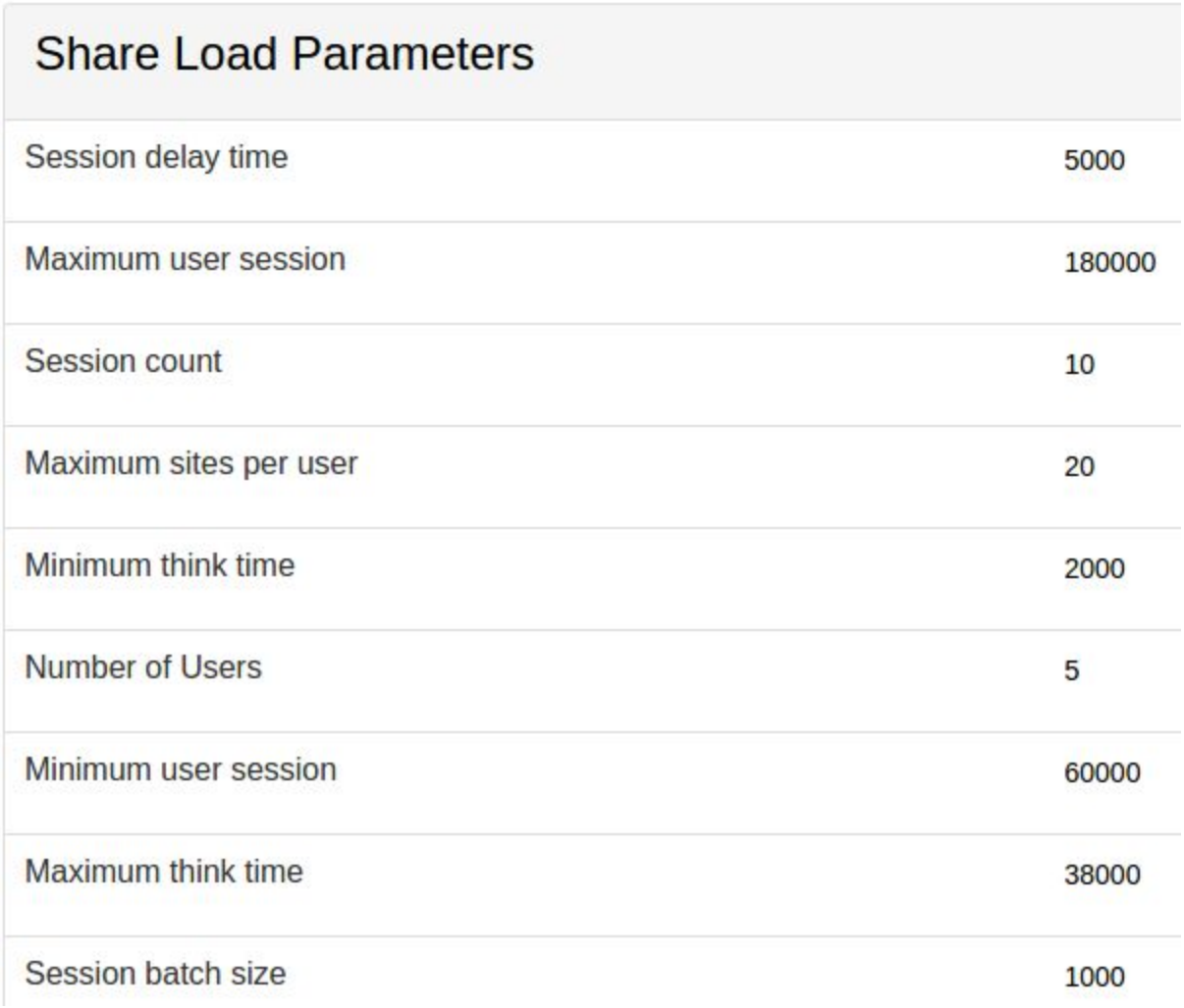
There are a number of other properties sections where we can configure stuff, most are standard and look the same for all load tests implementations. For example, the **Alfresco Server Details**:

This is where you set the connection information to use when communicating with the Alfresco Share server. If you want to test over a secure connection then change `Scheme` to `https` and `Alfresco port` to `8443`.

Note that it is important that the `Alfresco Server` hostname that is specified is the same hostname used by the Sign-Up test, so that the user data mirror name matches (which is `mirrors.localhost.users` from our previous Sign-Up test run). As mentioned before, this value can be set once at the test definition level when performing repeated runs against the same server. The hostname must be visible to the Driver server instances where the tests are deployed.

Note here also that the Alfresco Server Details property group for the Data Load test has more properties then we use here for the Share test. So you can pick and choose the properties you actually need when you implement a test.

Next we will look at the property groups that are specific to the Share test. The **Share Load Parameters** properties group can be used to control how users are concurrently accessing the system:



These properties can be used to determine the load on the Share user interface, they have the following meaning:

Property Name	**Default Value**	**Description**
Session delay time	5000 milliseconds	The time between each session start/creation.
Session count	10	The number of sessions to start/create.

Session batch size	1000	This is the maximum number of sessions that should be created before the Drivers can kick in. In our case with 10 sessions this does not have any effect. But let's say you want a session count of 100000, then you would not have to wait until all 100000 session have been created before the Drivers start working. They would start after the first session batch of 1000 have been created.
Minimum user session	60000 milliseconds	The minimum length of a user session.
Maximum user session	180000 milliseconds	The maximum length of a user session.
Maximum sites per user	20	This is the limit for how many sites a user can create when they get tasked randomly to create a site. This is just to control that a user is not randomly creating thousands of sites, as this is not very realistic to do manually.
Number of users	5	This does **not** actually dictate how many concurrent users you want to run with. It just says to the system that there needs to be at least 5 users created via the Sign-Up test. The test will fail if there are less than 5 users.
Minimum think time	2000 milliseconds	The minimum time between each user click in the Share UI.
Maximum think time	38000 milliseconds	The maximum time between each user click in the Share UI.

The Share users pause for between 2 seconds and 38 seconds (normal distribution; mean 20s) between each "click". These values, and the number of concurrent users, can be adjusted using the following formula:

`N = S/T` where:

- `N` = number of concurrent users (desired)
- `S` = mean user session time
- `T` = time between session starts (`Session delay time`)

So if we for example wanted to simulate 60 concurrent users load testing Share, then we can calculate the time between session starts as follows:

S = 120 seconds
N = 60 (desired number of users)
T = 120 / 60 = 2s

So this means updating the `Session delay time` property to `2000` milliseconds.
For the minimum or maximum number of users just use the minimum or maximum session times:

T = 4s
S_{min}=200s
S_{max}=280s
N_{min}=50
N_{max}=70

Clearly, the load on the system will increase if the user think time is decreased; but the speed of user clicks does not change the calculation of *concurrent users*.

While talking about concurrent users it is worth mentioning the **Events and Threads** property group. At first it might be tempting to try and configure and simulate concurrent users with these properties:

However, they are not used for this. These properties are used by the Benchmark Drivers when they process events and are sufficient for most hardware that you might be running the Driver servers on. If you have an extremely powerful Driver machine, then you can start looking at re-configuring these properties. They do not determine the rate of how a test is run, how each test is run is specific to the test.

There is another properties group that is specific to the Share test, and that is the **WebDrone** group:

WebDrone

WebDrone html max element wait time	12000
Page max render wait time	300000
Page popup max render time	500
WebDrone language setting	en
Selenium Grid URL	http://127.0.0.1:4444/wd/hub
Use 'RemoteFireFoxWebDrone' for a Selenium FireFox grid of 'FireFoxWebDrone' for development testing without a grid	RemoteFireFoxWebDrone
Wait time in second	1
Support download file types	application/octet-stream,text/plain,text/html,text/xmlc

The properties in this group are used to configure where the Driver servers can find the Selenium Grid hub that we started earlier. Default `Selenium Grid URL` will work in our case, but if you are running the Grid somewhere else update the hostname. We can also set values for how long we will wait for a web page to render.

When a Share test includes uploading a file to a folder this file is coming from an FTP site configured via the **Test Files** properties group:

Test Files

Download Directory Location where files can be written during download testing.	${java.io.tmpdir}/bm/downloaded-files
FTP Path Path on remote server containing test files	/sites/www.linuxfromscratch.org/images
FTP Server Hostname	ftp.mirrorservice.org
FTP Password	*******
Name of the file on the FTP server containing searches to perform (one per line)	searchterms.txt
FTP Port	21
FTP Username	anonymous
Test File Cache Path to location where test content will be cached for quick access.	${java.io.tmpdir}/bm/cached-files

We don’t need to change anything here, we just use the default FTP download site. The FTP site is used by the Drivers for loading files, downloads names from the FTP, and talking to the

database about what files it has for indexing. You can use your own FTP and fill it up with your specific content files for testing your specific solution.

The next properties group that we are going to look at is also specific to the Share test suite and is called **Share Scenario Parameters**:

Share Scenario Parameters	
Select dashboard from site dashboard scenario weight value	005
Select create site from dashboard scenario weight value	005
Download document from document details scenario weight	30
Select people from dashboard scenario weight value	005
People finder prefix search scenario weight value	95
Select repository from dashboard scenario weight value	000
Document details select dashboard scenario weight value	10
Document details upload new version scenario weight value	10
Repeat search scenario weight value	020
Select document library from site dashboard scenario weight value	090

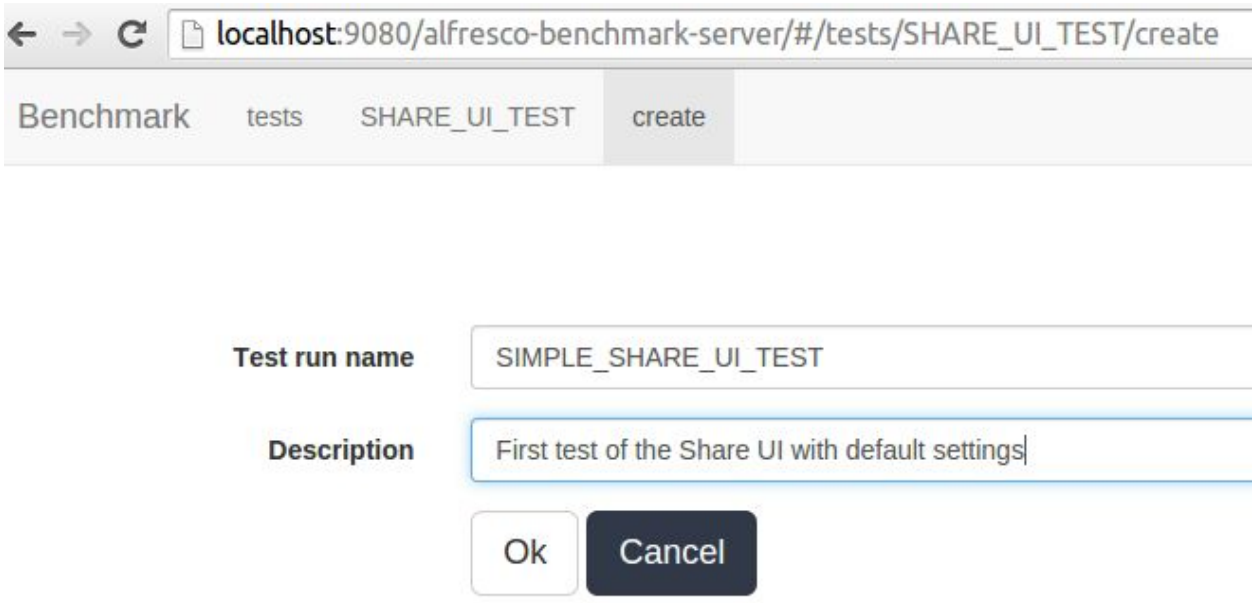
Note. the screenshot does not contain the complete list of parameters.

These properties control the importance (weight) of each individual test, making it easy for us to customize the Share user interface test according to client requirements. For example, if the solution we are building is not including any requirements for downloading files via Share, then we can set the “Download document from document details scenario weight” property to 0. This will effectively turn off this test so it is never executed.

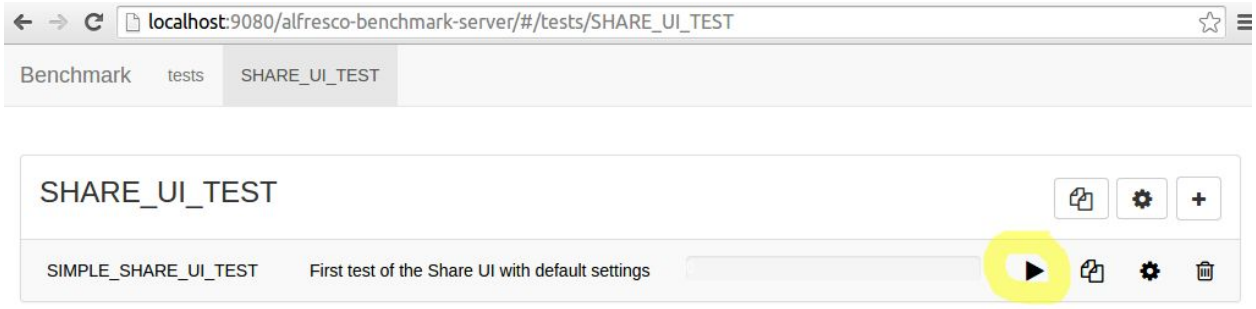
On the other hand, if our solution have loads of requirements where users create folders, then it might be useful to up the weight number for the “Create folder scenario weight value” property from 10 to for example 200. And so on.

Now to run a test based on this load test configuration we need to create a test run. We do this by clicking on the **+** at the top of the screen, just under the summary and description of the test:

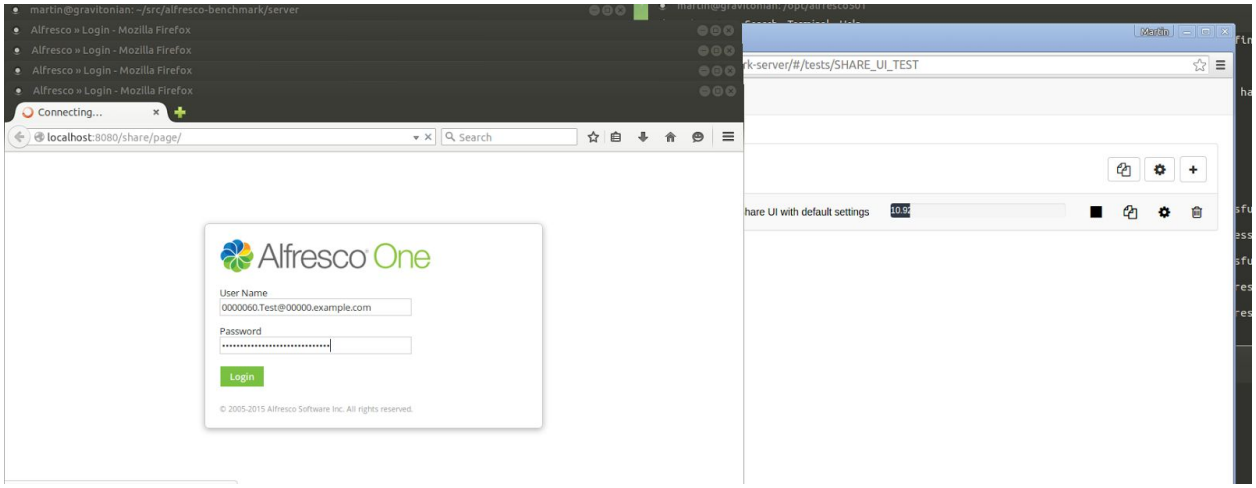
This presents the following screen where we can give the test run a name and description:



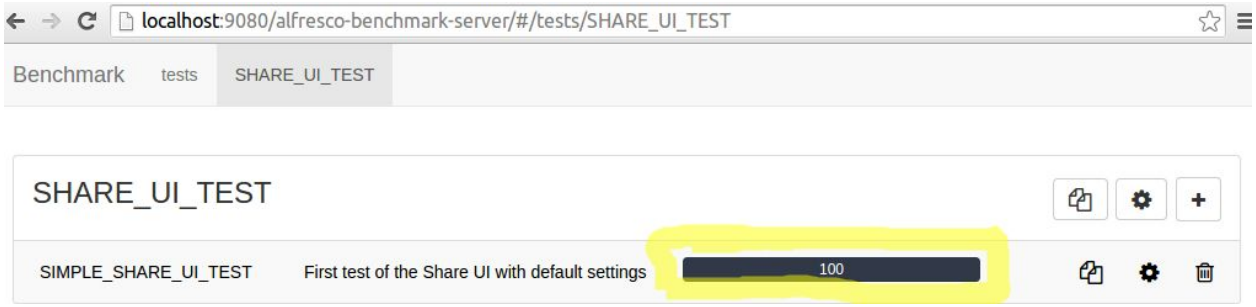
Note that the Test Definition name and the Test Run name makes up the name of the MongoDB collection where the Test Data will be stored, in this case `bm20-data/Collections/SHARE_UI_TEST.SIMPLE_SHARE_UI_TEST.*`. Click **OK** and you will see the screen where the test runs can be started:



Click the arrow to start the load test, it will be executed by both of the Benchmark Drivers that we started. You should see Firefox browsers starting to popup and the progress bar moving as in the following screenshot:



When the test is finished you will see the following screen:



Here you can click on the test row to navigate to a screen where you can download the load test result as a CSV or Excel file:

This completes the whole process of running the Share test from the Benchmark Management server.

Getting Started Writing Your Own Tests

Now when we have run through all these out-of-the-box tests wouldn't it also be interesting to see how we can create our own load tests. Quite often Alfresco projects require you to write your own custom Web Scripts. We will have a look at how you can test those with the Benchmark Toolkit.

Source code is available from here.

The All-in-One (AIO) project in the Alfresco SDK contains a Repository Web Script called Hello World that we can test. See these instructions for how to generate an AIO project.
The Hello World Web Script is quite simple so we are going to update it with a parameter representing a message that we are sending. After generating the AIO project update the Hello World Web Script as follows:

Descriptor (all-in-one/repo-amp/src/main/amp/config/alfresco/extension/templates/webscripts/helloworld.get.desc.xml):

```
<webscript>
  <shortname>Hello World Sample Webscript</shortname>
```

```
  <description>Hands back a greeting</description>
  <url>/sample/helloworld?message={message}</url>
  <authentication>user</authentication>
  <format default="html"></format>
</webscript>
```

Controller (all-in-one/repo-amp/src/main/amp/config/alfresco/extension/templates/webscripts/helloworld.get.js):

```
var messageParamName = "message";
var message = args[messageParamName];
model["fromJS"] = "Hello " + message;
```

Template (all-in-one/repo-amp/src/main/amp/config/alfresco/extension/templates/webscripts/helloworld.get.html.ftl):

```
Message from JS  : ${fromJS}
Message from Java: ${fromJava}
```

Pre-requisites

Before starting on implementing custom benchmark tests be sure to read through the Concepts section again as it is quite vital to understanding how it all works. It will also be very helpful if you are familiar with running one of the out-of-the-box tests, so if you have not yet done that, make sure to at least run the Sign-Up test before moving on. This test is actually required to have been run as we will use the Alfresco user accounts created by it when invoking the Web Script and authenticating.
Note that when you run the Sign-Up test you need to have the AIO project running by doing something like this:

```
all-in-one$ mvn clean install -Prun
```

This is so all the users are created in the correct Alfresco Repository.

Introduction

Load tests are implemented in Java and packaged as WAR files. When you write a new test the Benchmark Framework Services and ReST-based API are all inherited (see figure in the introduction). By far the easiest way to get going with writing your own custom test is to use a Sample test project generated from a Maven archetype.

This template project will contain the Sample load test that we tried out in the beginning. When we get an idea of how it works we can replace the sample load test with our own test.

The process for implementing and running a custom load test looks something like this:

1. Use the Maven archetype to generate a Maven project representing the test, it will already come with a sample test for reference
2. Replace the sample test code with your custom test code (i.e. testing the Repo Web Script)
3. Update the properties file to match needed properties for the test
4. Update the `app.properties` so the correct property groups are specified
5. Update the Spring context file `test-context.xml` so it matches the test flow
6. Run the unit test (`mvn install`), which will verify the generated code against an embedded MongoDB instance, it will also give you an easy way to debug the load test
7. Start up the maven project as a Benchmark Driver: `mvn tomcat7:run -Dmongo.config.host=localhost`
8. Download the Benchmark Management Server project and start it up and create a test and test run to try out the custom load test

Generate a sample load test project based on Maven Archetype

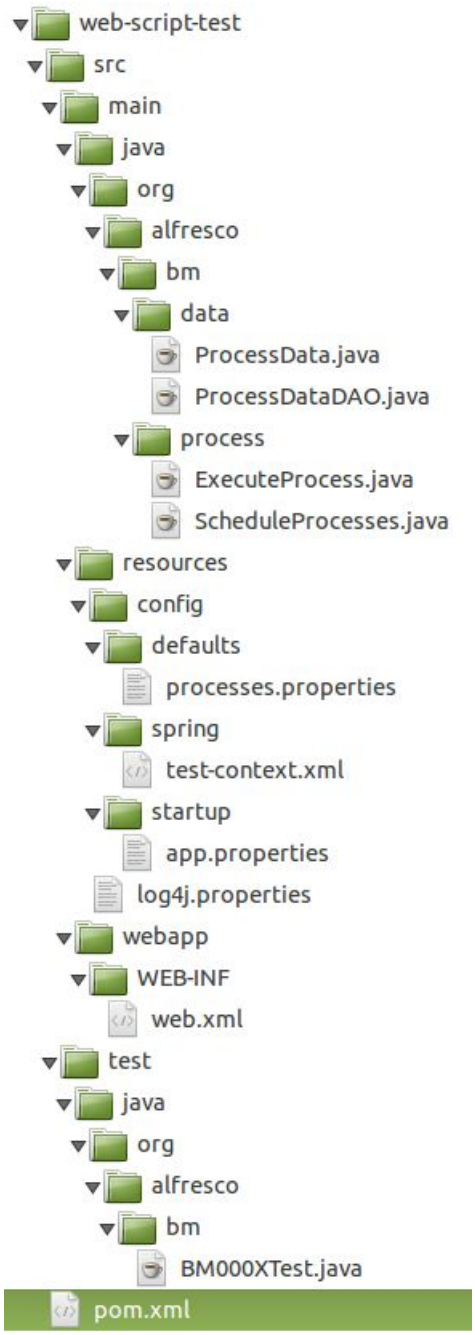
To generate the sample test project we use a Maven archetype as follows:

```
martin@gravitonian:~/src$ mvn archetype:generate -DgroupId=com.mycompany
-DartifactId=benchmark-web-script-test -DarchetypeGroupId=org.alfresco
-DarchetypeArtifactId=alfresco-benchmark-sample-archetype
-DarchetypeVersion=2.0.9 -DinteractiveMode=false^C
martin@gravitonian:~/src$ cd benchmark-web-script-test/
martin@gravitonian:~/src/benchmark-web-script-test$ ls -l
total 8
-rw-rw-r-- 1 martin martin 3284 Jun 17 15:15 pom.xml
drwxrwxr-x 4 martin martin 4096 Jun 17 15:15 src
```

Change the `groupId` and `artifactId` for your test as you like. Note also that there might be a newer version than `2.0.9` available for the artifact when you read this. If there is a newer version available then update the `archetypeVersion` property.

The sample load test project structure

The generated load test project will have a project structure looking something like this:



So what we got here is a sample load test implementation that we can run and test immediately. The files in the project have the following meaning:

- **`ProcessData.java`** - represents the event data used by the workflow process test, will be stored in MongoDB for later access
- **`ProcessDataDAO.java`** - this is a data access object that is used to persist and look for process data in MongoDB

- **ExecuteProcess.java** - Event processor implementation for executing a scheduled process (uses the timer, fetches event data from MongoDB, simulates executing a process)
- **ScheduleProcesses.java** - Event processor implementation for schduling processes that should be executed (writes process data to MongoDB including setting state to scheduled, creates session)
- **processes.properties** - These are test property definitions related to this load test(s). They control the load test execution, such as number of processes, delay between executing each process etc. Every property file under **resources/config/defaults** will be read, the name of the file is not important more than that it could be named after what you are testing. The user interface will be dynamically generated based on these property definitions. Property Naming Convention: `namespace.name.metadata`, name is injected into spring context (e.g. `proc.processCount`).
- **`test-context.xml`** - entry point into Spring context for load test component definitions (i.e. definitions of events processors, event producers, test file locations etc) - this context filename is hard-coded, it needs to have exactly this name.
- **app.properties** - Load test description and what properties that should be available. This filename is hard-coded. The `app.release` property is the test name and version. The benchmark libraries that are brought in via dependencies contain property files in the https://github.com/AlfrescoBenchmark/alfresco-benchmark/tree/master/server/src/main/resources/config/defaults directory. So when we specify `app.inheritance=SAMPLE,COMMON,FILES,FILES_FTP,FILES_LOCAL`, it means that we want our test to use all these property definitions. The `app.schema` property is used during development when you make a lot of test property changes, separate in config db, when releasing you could set to 0.
- **BM000XTest.java** - Unit test that is used to validate the load test, can also be useful for debugging the load test.
- **`pom.xml`** - maven project file that brings in the Benchmark Web Application implementation, including the ReST API, Services, and common property definitions. If you are going to deploy several Benchmark Drivers on a single host, serving different load tests or the same load test, then you need to change the port number in this file so it does not clash between the Driver instances. This file also contains the version number for the Benchmark framework that you are currently using.

Note here that this sample load test does not use any Alfresco server at all. You can do `mvn install` and run through it and it will not complain if there is no Alfresco server running. This sample load test just simulates executing fictive processes.

Replacing the sample load test with a Web Script load test

Before we can replace the sample test we need to figure out what we are going to test and how. We want to be able to configure how many times the Web Script should be invoked, with what

frequency, and what message to send. It should also be possible to configure what user that should be used when authenticating the call.
What this means is that we need test specific properties for things like invocation count and invocation delay.

The sequence of events is something like this:

1) Have a component (i.e. event processor) schedule x number of Web Script invocations
2) Have another component (i.e. event processor) make the invocations

Update the Sample Event Processors so they do the Web Script Test

The sample load test has an event processor called `ScheduleProcesses`. Rename it to `ScheduleWebScriptInvocationsEventProcessor`. Rename also the package it is stored in from `process` to `invokewebscript`. This event processor will be responsible for generating and scheduling events for all the Web Script invocations that we want to do. Update the event processor code so it looks like this:

```
package org.alfresco.bm.invokewebscript;

import java.util.ArrayList;
import java.util.List;
import java.util.UUID;

import org.alfresco.bm.data.DataCreationState;
import org.alfresco.bm.data.WebScriptInvocationData;
import org.alfresco.bm.data.WebScriptInvocationDataDAO;
import org.alfresco.bm.event.AbstractEventProcessor;
import org.alfresco.bm.event.Event;
import org.alfresco.bm.event.EventResult;
import org.alfresco.bm.user.UserDataService;
import org.apache.commons.logging.Log;
import org.apache.commons.logging.LogFactory;

public class ScheduleWebScriptInvocationsEventProcessor extends AbstractEventProcessor {
  private static Log logger = LogFactory.getLog(ScheduleWebScriptInvocationsEventProcessor.class);

  public static final int DEFAULT_BATCH_SIZE = 100;
  public static final String EVENT_NAME_WEB_SCRIPT_INVOCATION = "webScriptInvocation";

  private final UserDataService userDataService;
  private final WebScriptInvocationDataDAO webScriptInvocationDataDAO;
  private final String testRunFqn;
  private long numberOfWebScriptInvocations;
  private final long timeBetweenWebScriptInvocations;
  private String webScriptMessagePattern;
  private String eventNameWebScriptInvocation;
  private int batchSize;
```

```
public ScheduleWebScriptInvocationsEventProcessor(UserDataService userDataService,
      WebScriptInvocationDataDAO webScriptInvocationDataDAO,
      String testRunFqn, int numberOfWebScriptInvocations, long timeBetweenWebScriptInvocations,
                        String webScriptMessagePattern) {
   super();
   this.userDataService = userDataService;
   this.webScriptInvocationDataDAO = webScriptInvocationDataDAO;
   this.testRunFqn = testRunFqn;
   this.numberOfWebScriptInvocations = numberOfWebScriptInvocations;
   this.timeBetweenWebScriptInvocations = timeBetweenWebScriptInvocations;
   this.batchSize = DEFAULT_BATCH_SIZE;
   this.eventNameWebScriptInvocation = EVENT_NAME_WEB_SCRIPT_INVOCATION;
   this.webScriptMessagePattern = webScriptMessagePattern;
}

public void setBatchSize(int batchSize) {
   this.batchSize = batchSize;
}

@Override
public EventResult processEvent(Event event) throws Exception {
   // Check how many Web Script invocations that have already been scheduled.
   // This depends on the batchSize and the total number of invocations that should be made.
   Integer alreadyScheduled = (Integer) event.getData();
   if (alreadyScheduled == null) {
      alreadyScheduled = Integer.valueOf(0);
   }
   if (logger.isDebugEnabled()) {
      logger.debug("Already scheduled " + alreadyScheduled + " " + eventNameWebScriptInvocation +
            " events and will schedule up to " + batchSize + " more.");
   }

   // Schedule another batch of Web Script Invocation events
   List<Event> events = new ArrayList<Event>(batchSize + 1);
   long now = System.currentTimeMillis();
   long scheduled = now;
   int localCount = 0;
   int totalCount = (int) alreadyScheduled;
   for (int i = 0; i < batchSize && totalCount < numberOfWebScriptInvocations; i++) {
      // Create a unique name for this Web Script invocation and store it under this name in the MongoDB
      String webScriptInvocationName = testRunFqn + "-" + UUID.randomUUID();

      // Delay the invocation with specified time
      scheduled += timeBetweenWebScriptInvocations;

      // Store this Web Script invocation as Scheduled
      WebScriptInvocationData data = new WebScriptInvocationData();
      data.setName(webScriptInvocationName);
      String message = webScriptMessagePattern;
      if (message.contains("%")) {
```

```
            message = String.format(webScriptMessagePattern, totalCount);
        }
        data.setMessage(message);
        data.setUsername(userDataService.getRandomUser().getUsername());
        data.setState(DataCreationState.Scheduled);
        webScriptInvocationDataDAO.createWebScriptInvocation(data);

        // Attach Web Script Invocation name as the event data, so we can look up the event data from
        // other Event Processors
        Event webScriptInvocationEvent = new Event(eventNameWebScriptInvocation, scheduled,
webScriptInvocationName);

        // Add the Web Script Invocation event to the list of events scheduled
        events.add(webScriptInvocationEvent);
        localCount++;
        totalCount++;
    }

    // If we have not yet scheduled all the Web Script Invocations that we want to do, then reschedule this event
    if (totalCount < numberOfWebScriptInvocations) {
        Event rescheduleEvent = new Event(event.getName(), scheduled, Integer.valueOf(totalCount));
        events.add(rescheduleEvent);
    }

    // The ResultBarrier will ensure that this gets rescheduled, if necessary
    EventResult result = new EventResult("Created " + totalCount + " scheduled Web Script Invocations.",
events);

    // Done
    if (logger.isDebugEnabled()) {
        logger.debug("Scheduled " + localCount + " Web Script Invocations and " + (totalCount <
numberOfWebScriptInvocations ?
                "rescheduled" : "did not reschedule") + " self.");
    }

    return result;
  }
}
```

A standard event processor should always extend the `AbstractEventProcessor` class. It will automatically register an event name with the Benchmark framework so it knows when to kick it off. The event name is taken from the Spring bean `id` that is used for the bean defining the event processor component, which in this case is (see `src/main/resources/config/spring/test-context.xml`):

```
<bean id="event.scheduleWebScriptInvocations"
```

The event name is the last bit after "`event.`", so in this case it will be `scheduleWebScriptInvocations`. The `AbstractEventProcessor` class also has a stop watch built in that we can use when timing the event processing.

The member variables of this event processor have the following meaning:

Variable name	Description
userDataService	*User data service to get hold of usernames to use for Web Script invocation authentication. There must be some users created in Alfresco (and in the mirror) for this to work. This can be done via the Sign-Up test.*
webScriptInvocationDataDAO	*Web Script Invocation data access object. This is a custom DAO just for this test that will have methods for storing and retrieving Web Script Invocation data.*
testRunFqn	*Fully qualified Name (FQN) for the active Test Run. This is the name you give it in the Benchmark Management UI.*
numberOfWebScriptInvocations	*Total number of Web Script invocations that should be executed (property is specific to this test)*
timeBetweenWebScriptInvocations	*The delay (millisec) between each Web Script invocation (property is specific to this test).*
webScriptMessagePattern	*A pattern for how the generated Web Script message parameter value should look like.* *For example:* *"Message 0000001"* *"Message 0000002"* *etc.*
eventNameWebScriptInvocation	*The event name for a Web Script Invocation, will trigger an event processor that calls the Web Script.*
batchSize	*The number of Web Script invocations that should be scheduled in one go. For example, if we should make in total 500 Web Script invocations, then we can set the batch size to 100, and then have the invocations starting before all have been scheduled.*

The event processor implementation starts out by checking how many Web Script invocations that have already been scheduled. The `ScheduleWebScriptInvocations` event carries the number of already scheduled invocations as data so we can check it at the start of the `processEvent` method. The number of `webScriptInvocation` events scheduled per call is controlled by the `batchSize` property, which is 100 by default. The reason we want to schedule the Web Script Invocation events in batches is so multiple drivers supporting this test can start processing the events as soon as we reach batch size.

For each `webScriptInvocation` event that is created we set its name to the “Test Run name + UUID”. We then save all the Web Script invocation data keyed on this name. The data is things such as Web Script message parameter pattern, username, and state. The data is saved to the MongoDB data mirror named after the `webscript.invocations.collection.name` property. This Web Script Invocation name is also sent as event data to the `webScriptInvocation` event, so we can easily get to the data from its event processor implementation.

At the end of the `processEvent` method we check if there are more `webScriptInvocation` events that should be scheduled, if so we reschedule ourselves, which means that the `ScheduleWebScriptInvocationsEventProcessor` will be called again and schedule up to another 100 invocations. The event result sent from this event processor will be a message saying how many `webScriptInvocation` events that were scheduled and a list of all those events.

The sample load test also has an event processor called Execute`Process`. Rename it to `InvokeWebScriptEventProcessor`. This event processor will be responsible for doing the actual Web Script call/invocation. Update the event processor code so it looks like this:

```
package org.alfresco.bm.invokewebscript;

import java.net.URLEncoder;
import java.util.Collections;

import org.alfresco.bm.data.DataCreationState;
import org.alfresco.bm.data.WebScriptInvocationData;
import org.alfresco.bm.data.WebScriptInvocationDataDAO;
import org.alfresco.bm.event.Event;
import org.alfresco.bm.event.EventResult;
import org.alfresco.bm.http.AuthenticatedHttpEventProcessor;

import org.alfresco.bm.user.UserData;
import org.alfresco.bm.user.UserDataService;
import org.alfresco.http.AuthenticationDetailsProvider;
import org.alfresco.http.HttpClientProvider;
import org.alfresco.http.SimpleHttpRequestCallback;
import org.apache.http.HttpResponse;
import org.apache.http.HttpStatus;
import org.apache.http.StatusLine;
import org.apache.http.client.methods.HttpGet;

public class InvokeWebScriptEventProcessor extends AuthenticatedHttpEventProcessor {
  public static final String EVENT_NAME_WEB_SCRIPT_INVOCATION_DONE = "webScriptInvocationDone";
  private static final String HELLO_WORLD_WS_URL = "/alfresco/service/sample/helloworld?message=";

  private final UserDataService userDataService;
  private final WebScriptInvocationDataDAO webScriptInvocationDataDAO;
```

```
  private String eventNameWebScriptInvocationDone;

  public InvokeWebScriptEventProcessor(
        HttpClientProvider httpClientProvider,
        AuthenticationDetailsProvider authenticationDetailsProvider,
        String baseUrl,
        WebScriptInvocationDataDAO webScriptInvocationDataDAO,
        UserDataService userDataService) {
    super(httpClientProvider, authenticationDetailsProvider, baseUrl);
    this.userDataService = userDataService;
    this.webScriptInvocationDataDAO = webScriptInvocationDataDAO;
    this.eventNameWebScriptInvocationDone = EVENT_NAME_WEB_SCRIPT_INVOCATION_DONE;
  }

  @Override
  public EventResult processEvent(Event event) throws Exception {
    // Usually, the entire method is timed but we can choose to control this
    super.suspendTimer();

    // Get the Web Script Invocation name
    String webScriptInvocationName = (String) event.getData();

    // Locate the Web Script Invocation data and make a quick check on it
    WebScriptInvocationData webScriptInvocationData =
          webScriptInvocationDataDAO.findWebScriptInvocationByName(webScriptInvocationName);
    EventResult result = null;
    if (webScriptInvocationData == null) {
      result = new EventResult(
            "Skipping processing for '" + webScriptInvocationName + "'.  Web Script Invocation data not
found.",
            false);
      return result;
    } else if (webScriptInvocationData.getState() != DataCreationState.Scheduled) {
      result = new EventResult(
            "Skipping processing for '" + webScriptInvocationName + "'.  Web Script Invocation not
scheduled.",
            false);
      return result;
    } else if (webScriptInvocationData.getUsername() == null) {
      result = new EventResult(
            "Skipping processing for '" + webScriptInvocationName + "'.  Web Script Invocation has no
username.",
            false);
      return result;
    } else if (webScriptInvocationData.getMessage() == null) {
      result = new EventResult(
            "Skipping processing for '" + webScriptInvocationName + "'.  Web Script Invocation has no
message.",
            false);
      return result;
```

```
}

EventResult eventResult = null;

// Look up the user data for the username that will be used to authenticate and invoke the Web Script
UserData user = userDataService.findUserByUsername(webScriptInvocationData.getUsername());
if (user == null) {
   eventResult = new EventResult("User data not found in local database: " +
         webScriptInvocationData.getUsername(), Collections.EMPTY_LIST,
         false);
   return eventResult;
}

// Start the clock that times the Web Script call
resumeTimer();

// Make the Web Script call authenticated as username
// WebScript Call will have a URL looking something like:
//    http://localhost:8080/alfresco/service/sample/helloworld?message=Message%200000003
HttpGet webScriptInvocationGet = new HttpGet(
      getFullUrlForPath(HELLO_WORLD_WS_URL +
            URLEncoder.encode(webScriptInvocationData.getMessage(), "UTF-8")));
HttpResponse httpResponse =
      executeHttpMethodAsUser(webScriptInvocationGet, webScriptInvocationData.getUsername(),
      SimpleHttpRequestCallback.getInstance());
StatusLine httpStatus = httpResponse.getStatusLine();

// Stop the clock, we are done with the Web Script call
suspendTimer();

// Check if the Alfresco server responded with OK
if (httpStatus.getStatusCode() == HttpStatus.SC_OK) {
   // Record the name of the Web Script Invocation to reflect that is was executed on the Alfresco server
   boolean updated = webScriptInvocationDataDAO.updateWebScriptInvacationState(
         webScriptInvocationName, DataCreationState.Created);
   if (updated) {
      // Create 'done' event, which will not have any further associated event processors
      Event doneEvent = new Event(eventNameWebScriptInvocationDone, 0L, webScriptInvocationName);
      eventResult = new EventResult("Web Script Invocation " + webScriptInvocationName +
            " completed.", doneEvent);
   } else {
      throw new RuntimeException("Web Script Invocation " + webScriptInvocationName +
            " was executed but not recorded.");
   }
} else {
   // Web Script Invocation failed
   String msg = String.format("Web Script call failed, ReST-call resulted in status:%d with error %s ",
         httpStatus.getStatusCode(), httpStatus.getReasonPhrase());
   eventResult = new EventResult(msg, Collections.<Event>emptyList(), false);
```

```
            webScriptInvocationDataDAO.updateWebScriptInvacationState(webScriptInvocationName,
DataCreationState.Failed);
        }

        return eventResult;
    }
}
```

This event processor is going to make ReST calls so we know we are going to need some component that can make HTTP calls. The Benchmark Toolkit already got an event processor base called `AuthenticatedHttpEventProcessor` that we can use, it has functionality to make authenticated HTTP calls. This event processor extends the `AbstractEventProcessor` class so we get the basic event registration functionality too.

The `AuthenticatedHttpEventProcessor` constructor takes a HTTP Client Provider, Authentication Details Provider, and a base URL. So our event processor constructor needs to have these objects passed in too, we will see when we discuss the Spring context how this is done. The constructor also needs a Web Script Invocation DAO passed in so we can get to the Web Script invocation data for the event that should be processed.

The event processor implementation starts of with getting the event data, which contains the Web Script Invocation name that we constructed in the `ScheduleWebScriptInvocationsEventProcessor`. With the Web Script Invocation name we can get to the rest of the data via the `webScriptInvocationDataDAO`. After we get the data we check it so it is complete. If something is missing we return an `EventResult` with `success` set to `false` and an error message.

We then use the `userDataService` to get the Alfresco user account for the `username` that was fetched via the Web Script Invocation data. If there is no user we return an `EventResult` with `success` set to `false`. Remember, this test requires the Sign-Up test to have been run before so that we got some user accounts in the Alfresco Repository.

The next thing we do is setting up a HTTP Get call via the `org.apache.http.client.methods.HttpGet` class. To construct the Web Script URL we use the `AuthenticatedHttpEventProcessor.getFullUrlForPath` method to which we pass in the whole URL except hostname and port, which will be set to the `baseUrl`. We make sure to encode the URL parameter so spaces and other stuff in the parameter value are encoded.

Then we use the `AuthenticatedHttpEventProcessor.executeHttpMethodAsUser` method to make the authenticated Web Script call as the user passed in the Web Script Invocation data. After this there is the standard check of the HTTP response status to figure out if it was successful or not. In case of a successful call we update the data creation state to `Created`, which is a bit confusing as we did not actually create any node in the Repository, but

it represents a successful call. We also create an `EventResult` where we pass in the `eventNameWebScriptInvocationDone` event name, which will result in no more processing.

This is all the event processing implementation that we need, next is the implementation of the event data access object (DAO) and the event data transfer object (DTO).

Implementing the event DAO and DTO

These classes are quite standard and have nothing specific to do with the Benchmark Toolkit, it is more about how to insert *event* data into the MongoDB, how to get *event* data out of MongoDB, and how to transfer it around. Instead of copy and paste the classes here, download the source code and have a look at them (note which Sample load test class that have been renamed to what):

- `ProcessData` renamed to src/main/java/org/alfresco/bm/data/`WebScriptInvocationData`.java
- `ProcessDataDAO` renamed to src/main/java/org/alfresco/bm/data/`WebScriptInvocationDataDAO`.java

Defining the Spring test context

All the events that make up a test are tied together in a spring context loaded from the `src/main/resources/config/spring/test-context.xml` file. This file needs to be updated based on what we want to do in our Web Script test. Open up the file and change it so it starts out like this:

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://www.springframework.org/schema/beans
      http://www.springframework.org/schema/beans/spring-beans-3.0.xsd">

  <!-- Import any application contexts for test runs -->
  <import resource="classpath:config/spring/test-common-context.xml" />

  <!--               -->
  <!-- Reporting     -->
  <!--               -->

  <!-- The CompletionEstimator looking at the Web Script invocation count results -->
  <bean id="completionEstimator.webScriptInvocationCount"
class="org.alfresco.bm.test.EventCountCompletionEstimator">
    <constructor-arg name="eventService" ref="eventService" />
    <constructor-arg name="resultService" ref="resultService" />
    <constructor-arg name="eventName" value="webScriptInvocation" />
    <constructor-arg name="eventCount" value="${webscript.numberOfInvocations}" />
  </bean>
```

```
<!-- Override the default estimator to use the desired estimators -->
<bean id="completionEstimator" class="org.alfresco.bm.test.CompoundCompletionEstimator">
   <constructor-arg name="eventService" ref="eventService" />
   <constructor-arg name="resultService" ref="resultService" />
   <constructor-arg name="estimators">
      <list>
         <ref bean="completionEstimator.elapsedTime" />
         <ref bean="completionEstimator.webScriptInvocationCount" />
      </list>
   </constructor-arg>
</bean>
```

The first thing we do in our test context is bringing in a common test context with properties and test components that we can then use in our custom test definition.

We then use the out-of-the-box `EventCountCompletionEstimator` to get reporting on how many Web Script invocations that have been executed at a certain time. The `eventName` and `eventCount` is set so it knows what events to count and how many to expect. The `webscript.numberOfInvocations` property is specific to this test and we will define it in a bit. You can use the `CompoundCompletionEstimator` to get the highest estimate from a list of estimators.

Next thing we define in out test context is a number of support services:

```
<bean id="userDataService" class="org.alfresco.bm.user.UserDataServiceImpl">
  <constructor-arg name="db" ref="testMongoDB" />
  <constructor-arg name="collection" value="${users.collection.name}" />
</bean>

<bean id="authenticationDetailsProvider"
class="org.alfresco.bm.http.UserDataAutheticationDetailsProvider">
  <constructor-arg name="userDataService" ref="userDataService" />
  <constructor-arg name="adminUserName" value="${alfresco.adminUser}" />
  <constructor-arg name="adminPassword" value="${alfresco.adminPwd}" />
</bean>

<!-- Shared HTTP-Client instance provider to have better performance -->
<bean id="httpClientProvider" class="org.alfresco.http.SharedHttpClientProvider">
  <constructor-arg name="maxNumberOfConnections" value="${http.connection.max}" />
  <constructor-arg name="connectionTimeoutMs" value="${http.connection.timeoutMs}" />
  <constructor-arg name="socketTimeoutMs" value="${http.socket.timeoutMs}" />
  <constructor-arg name="socketTtlMs" value="${http.socket.ttlMs}" />
</bean>
```

The `userDataService` is used to access the user accounts that have been created with the Sign-Up test. They are stored in MongoDB in a collection with the name specified with the `users.collection.name` property.

Then we define an `authenticationDetailsProvider` that is going to help us authenticate the Web Script calls with the username passed in the Web Script Invocation data. The `userDataService` is used to get to the password for each user.

Finally we define a `httpClientProvider`, which is basically a HTTP Client from the Apache HTTP Components library that has been improved a bit, here's the JavaDocs for the `SharedHttpClientProvider`:

A class responsible for managing shared HttpClient HTTP connections. This uses a thread-safe connection-manager instead of creating a new instance on every call. This is done for the following reasons:

- Creating new `HTTPClient` instances for each request has memory-overhead and most important, opens a new connection for each request. Even though the connection is released they are kept in 'CLOSE_WAIT' state by the OS. This can, on high usage, empty out available outgoing TCP-ports and influence the testing on the client, having impact on the results.
- Using a single `HTTPClient` allows creating a pool of connections that can be kept alive to a certain route (route = combination of host and port) for a max number of time. This eliminates connection setup and additional server round-trips, raising the overall throughput of http-calls (web browsers use this mechanism all the time to lower loading-times).

The next thing we define is the Data Access Object that will be used to get to the test specific Web Script Invocation data:

```
<bean id="webScriptInvocationDataDAO" class="org.alfresco.bm.data.WebScriptInvocationDataDAO" >
  <constructor-arg name="db" ref="testMongoDB" />
  <constructor-arg name="collection" value="${webscript.invocations.collection.name}" />
</bean>
```

This data will be kept in a MongoDB collection with the name specified by the `webscript.invocations.collection.name` property.

The last thing to define in the test context is the event processors and event producers, which makes up the actual test flow:

```
<bean id="event.start" class="org.alfresco.bm.event.RenameEventProcessor" parent="event.base" >
  <constructor-arg name="outputEventName" value="scheduleWebScriptInvocations" />
  <property name="chart" value="false"/>
</bean>
```

```
<bean id="event.scheduleWebScriptInvocations"
class="org.alfresco.bm.invokewebscript.ScheduleWebScriptInvocationsEventProcessor"
parent="event.base" >
  <constructor-arg name="userDataService" ref="userDataService" />
  <constructor-arg name="webScriptInvocationDataDAO" ref="webScriptInvocationDataDAO" />
  <constructor-arg name="testRunFqn" value="${testRunFqn}" />
  <constructor-arg name="numberOfWebScriptInvocations" value="${wstest.numberOfInvocations}" />
  <constructor-arg name="timeBetweenWebScriptInvocations" value="${wstest.timeBetweenInvocations}" />
  <constructor-arg name="webScriptMessagePattern" value="${wstest.messageparam}" />
  <property name="batchSize" value="${wstest.scheduleBatchSize}" />
  <property name="chart" value="true" />
</bean>

<bean id="producer.webScriptInvocation" class="org.alfresco.bm.event.producer.RedirectEventProducer"
parent="producer.base" >
  <constructor-arg name="newEventName" value="invokeWebScript" />
  <constructor-arg name="delay" value="1" />
</bean>

<bean id="event.invokeWebScript"
class="org.alfresco.bm.invokewebscript.InvokeWebScriptEventProcessor" parent="event.base" >
  <constructor-arg name="httpClientProvider" ref="httpClientProvider" />
  <constructor-arg name="authenticationDetailsProvider" ref="authenticationDetailsProvider" />
  <constructor-arg name="baseUrl" value="${alfresco.url}" />
  <constructor-arg name="webScriptInvocationDataDAO" ref="webScriptInvocationDataDAO" />
  <constructor-arg name="userDataService" ref="userDataService" />
  <property name="chart" value="true" />
</bean>

<bean id="producer.webScriptInvocationDone"
class="org.alfresco.bm.event.producer.TerminateEventProducer" parent="producer.base" />
```

Every test starts with an event called `start`, which is a reserved event name that marks the start of a load test with a sequence of events that should be processed. A start event is only processed once so we cannot use this event name for something that can be processed more than once, such as for example our `scheduleWebScriptInvocations` event.

We set the implementation of the start event to the `RenameEventProcessor`, which basically just renames the `start` event to `scheduleWebScriptInvocations`, effectively kicking off the scheduling of all the Web Script invocations.

We feed the `scheduleWebScriptInvocations` event processor bean with both the user service and the DAO. We also set properties specifying how many Web Script invocations we want to do (i.e `wstest.numberOfInvocations`) and the delay between each one (i.e. `wstest.timeBetweenInvocations`). We also feed it with the Web Script Invocation parameter pattern (i.e. `wstest.messageparam`) so it knows how to construct the message

parameter in the Hello World Web Script call. Finally the batch size is set via the `wstest.scheduleBatchSize` property.

After all the Web Script Invocations have been scheduled, meaning the data for them have have been written to the MongoDB collection and the `webScriptInvocation` events for them generated and passed on via the `EventResult`, they will be picked up by the `RedirectEventProducer` that has been registered for the `webScriptInvocation` event. It will just redirect to the `invokeWebScript` event, which will inturn do the actual Web Script call via the `InvokeWebScriptEventProcessor`, which is registered on the `invokeWebScript` event. Using the `RedirectEventProducer` component is not really necessary in this test scenario, it just demonstrates that you can redirect an inbound event to some other event and delay the whole thing a bit if you want. We could have skipped it and just scheduled `invokeWebScript` event instead.

The `InvokeWebScriptEventProcessor` uses the `httpClientProvider` to make the Web Script HTTP call. It also uses the `authenticationDetailsProvider` to authenticate the call with the user provided in the Web Script Invocation data, which is accessed via the `webScriptInvocationDataDAO` . The `alfresco.url` property will contain the hostname and port for where the Alfresco Repository is running, and will make up the `baseUrl`.

When the `InvokeWebScriptEventProcessor` has finished processing an `invokeWebScript` event it creates a `webScriptInvocationDone` event, which will trigger a `TeminateEventProducer`, which in turn will terminate the processing.

We have seen quite a few properties being used when defining the beans that are part of the test. In the next section we will have a look at where to configure them.

Defining the test properties

We will do this in a file called `invoke-webscript.properties` (rename processes.properties). This file can be called whatever you like, as long as it is located in the`src/main/resources/config/defaults` directory of the project.

It will contain our custom test specific properties plus other more general properties for things like Alfresco server connection information:

```
# Alfresco Server Details

SERVER.alfresco.server.default=localhost
SERVER.alfresco.server.type=string
SERVER.alfresco.server.regex=[a-zA-Z0-9\.]*
SERVER.alfresco.server.group=Alfresco Server Details

SERVER.alfresco.port.default=8080
SERVER.alfresco.port.type=int
SERVER.alfresco.port.group=Alfresco Server Details
```

```
SERVER.alfresco.url.default=http://${alfresco.server}:${alfresco.port}/
SERVER.alfresco.url.type=string
SERVER.alfresco.url.group=Alfresco Server Details

SERVER.alfresco.adminUser.default=admin
SERVER.alfresco.adminUser.type=string
SERVER.alfresco.adminUser.mask=true
SERVER.alfresco.adminUser.group=Alfresco Server Details

SERVER.alfresco.adminPwd.default=admin
SERVER.alfresco.adminPwd.type=string
SERVER.alfresco.adminPwd.mask=true
SERVER.alfresco.adminPwd.group=Alfresco Server Details

# MongoDB Data Mirrors for the Alfresco Users and the Web Script invocation details

MIRROR.users.collection.name.default=mirrors.${alfresco.server}.users
MIRROR.users.collection.name.type=string
MIRROR.users.collection.name.title=User Data Mirror Name
MIRROR.users.collection.name.description=The name of a MongoDB collection to contain the user details.
The format is 'mirror.xyz.users'.
MIRROR.users.collection.name.group=Data Mirrors

MIRROR.webscript.invocations.collection.name.default=mirrors.${alfresco.server}.webscriptinvocations
MIRROR.webscript.invocations.collection.name.type=string
MIRROR.webscript.invocations.collection.name.regex=[a-zA-Z0-9]*
MIRROR.webscript.invocations.collection.name.title=Web Script Invocations Data Mirror Name
MIRROR.webscript.invocations.collection.name.description=The name of a MongoDB collection to contain
the Web Script Invocation details. The format is 'mirror.xyz.webscriptinvocations'.
MIRROR.webscript.invocations.collection.name.group=Data Mirrors

# Web Script Invocation Load Parameters

LOAD.wstest.numberOfInvocations.default=200
LOAD.wstest.numberOfInvocations.type=int
LOAD.wstest.numberOfInvocations.min=1
LOAD.wstest.numberOfInvocations.title=Number of Web Script invocations
LOAD.wstest.numberOfInvocations.group=Web Script Invocation Load Parameters

LOAD.wstest.timeBetweenInvocations.default=50
LOAD.wstest.timeBetweenInvocations.type=int
LOAD.wstest.timeBetweenInvocations.min=1
LOAD.wstest.timeBetweenInvocations.title=Web Script Invocation Delay
LOAD.wstest.timeBetweenInvocations.description=Milliseconds between each Web Script Invocation event
LOAD.wstest.timeBetweenInvocations.group=Web Script Invocation Load Parameters

# Internal
LOAD.wstest.scheduleBatchSize.default=100
LOAD.wstest.scheduleBatchSize.type=int
```

```
LOAD.wstest.scheduleBatchSize.hide=true
LOAD.wstest.scheduleBatchSize.group=Web Script Invocation Load Parameters

# Web Script Invocation Details

WSINVOCATION.wstest.messageparam.default=Message %07d
WSINVOCATION.wstest.messageparam.type=string
WSINVOCATION.wstest.messageparam.title=Message Parameter
WSINVOCATION.wstest.messageparam.description=The pattern for the Web Script invocation parameter
'message'.\n'Message %07d' will give 'Message 0000001' for the first message.
WSINVOCATION.wstest.messageparam.group=Web Script Invocation Details
```

The properties file specifies a number of different groups of properties, `SERVER`, `MIRROR`, `LOAD`, and `WSINVOCATION`. The grouping is so they can be displayed in these groups in the UI. You can also control if you want to include the group in the UI or not via its name. A group is also referred to as a namespace. Group names can be whatever you like but there is a certain convention in the out-of-the-box tests that can be worth following, such as SERVER, MIRROR, and LOAD.

The user interface will be dynamically generated based on these property definitions. The property naming convention is "`{namespace}.{property name}.{metadata}`". The property name is injected into spring context (e.g. `wstest.messageparam`).

The `SERVER` group specifies where the Alfresco Repository server is running and what the admin username and password is, if we were going to make any calls as admin user. The LOAD group specifies most of the properties needed to run the test, such as number of invocations we want to do. The WSINVOCATION group contains just one very test specific property that defines the pattern for the `message` parameter of the Web Script URL.

The way you control what properties are available in the UI is by the `app.properties` file located in the `src/main/resources/config/startup` directory of the project. It has the following property where the property groups are configured:

```
app.inheritance=COMMON,SERVER,HTTP,MIRROR,LOAD,WSINVOCATION
```

Note the property group called `COMMON`, it is not specified in our properties file but instead in the out-of-the-box properties files located in the `alfresco-benchmark/server/src/main/resources/config/defaults` directory. These properties control threading, MongoDB connection, and test duration configuration.

The `app.properties` file also got another important property called `app.schema`, it defines a specific test definition that has been deployed and stored in MongoDB. If you change the test properties and then build and restart the Driver (i.e. the test WAR), the changes will not be recognized as the schema has not been updated. In this case you need to increase the schema version before building and trying again.

Running the Web Script Invocation Load Test

We are now ready to give the test a go, everything is implemented and configured. The Benchmark Management Server need to be started, if it is not see here how to do it. Then the Alfresco server need to be started, which is done from the All-In-One SDK project that has the updated Hello World Web Script:

```
all-in-one$ mvn clean install -Prun
```

The Driver for the Web Script Invocation test is then started like this::

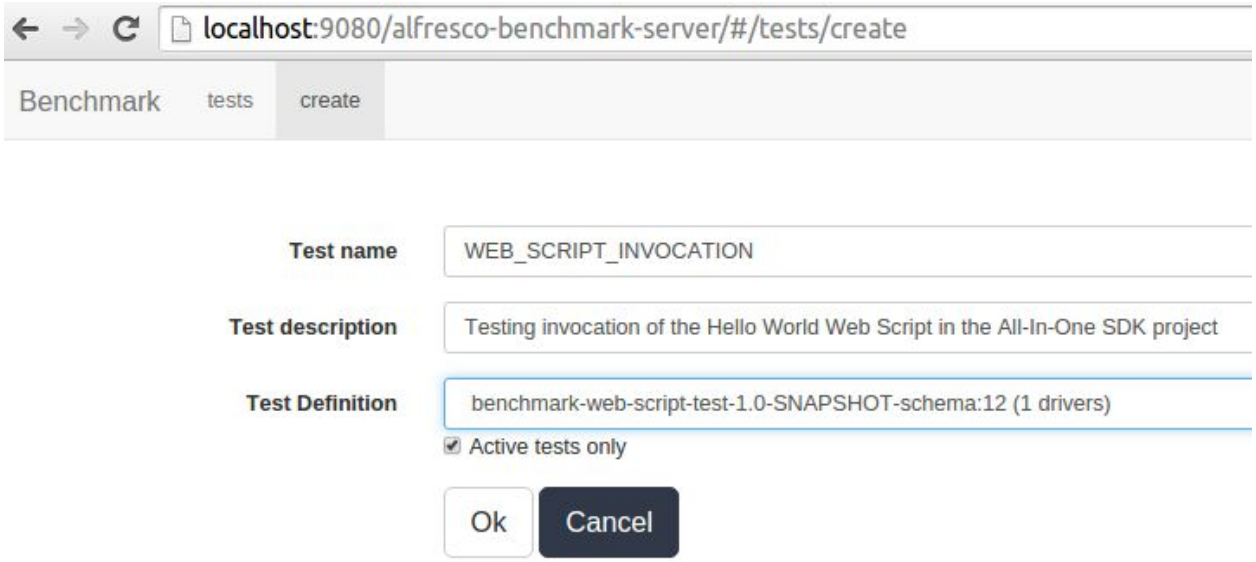
```
benchmark-web-script-test$ mvn clean tomcat7:run -Dmongo.config.host=localhost
```

This also produces the WAR with the test as follows:

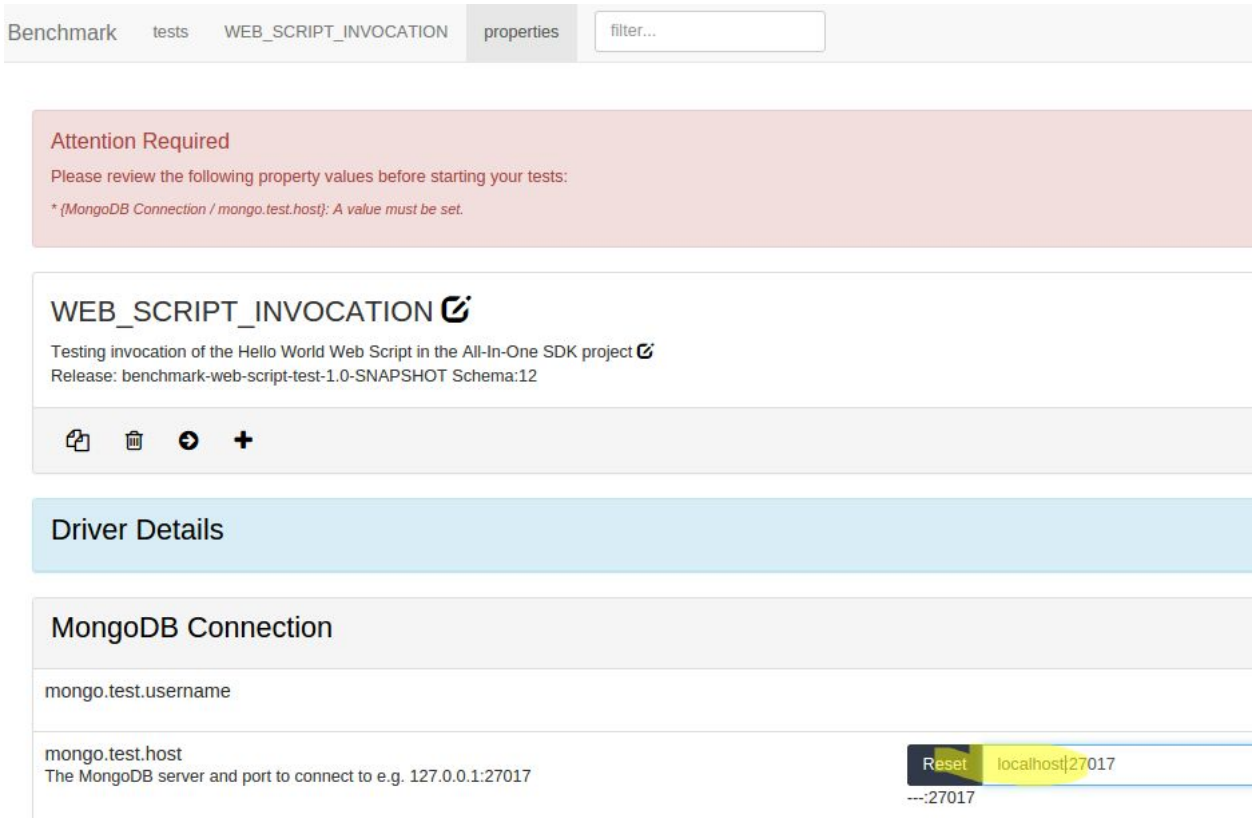
```
benchmark-web-script-test/target/benchmark-web-script-test-1.0-SNAPSHOT.war
```

This WAR file can also be deployed into a stand-alone Tomcat server and act as a Benchmark Driver for this load test.

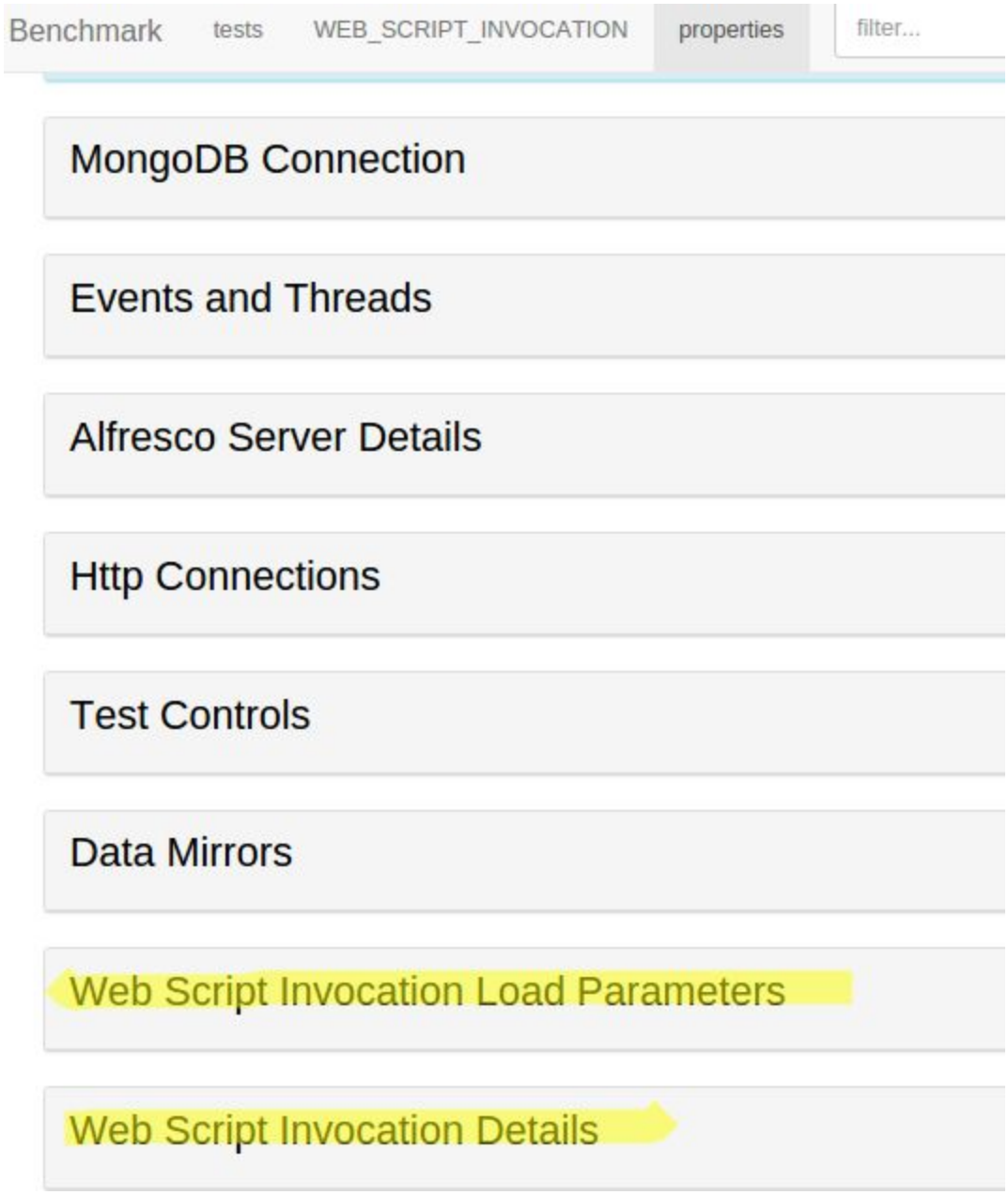
Now, access the Benchmark Management UI vi the http://localhost:9080/alfresco-benchmark-server URL and then click the big **+** to add another test definition:



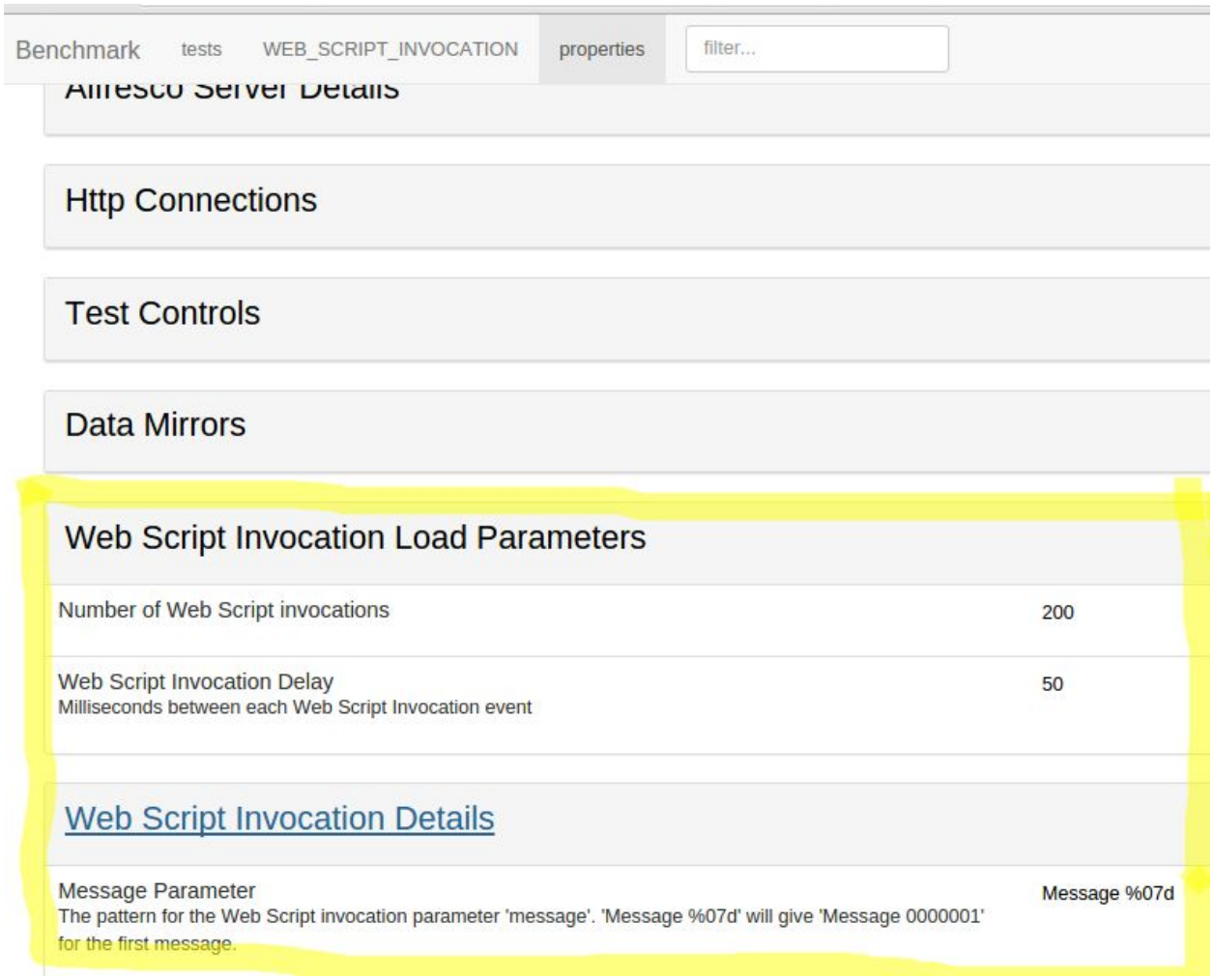
Click **Ok** when you are done filling in the fields and selecting test definition, there can be multiple test definitions if you have updated the properties and schemas between runs, pick the latest one. The following screen is now displayed:



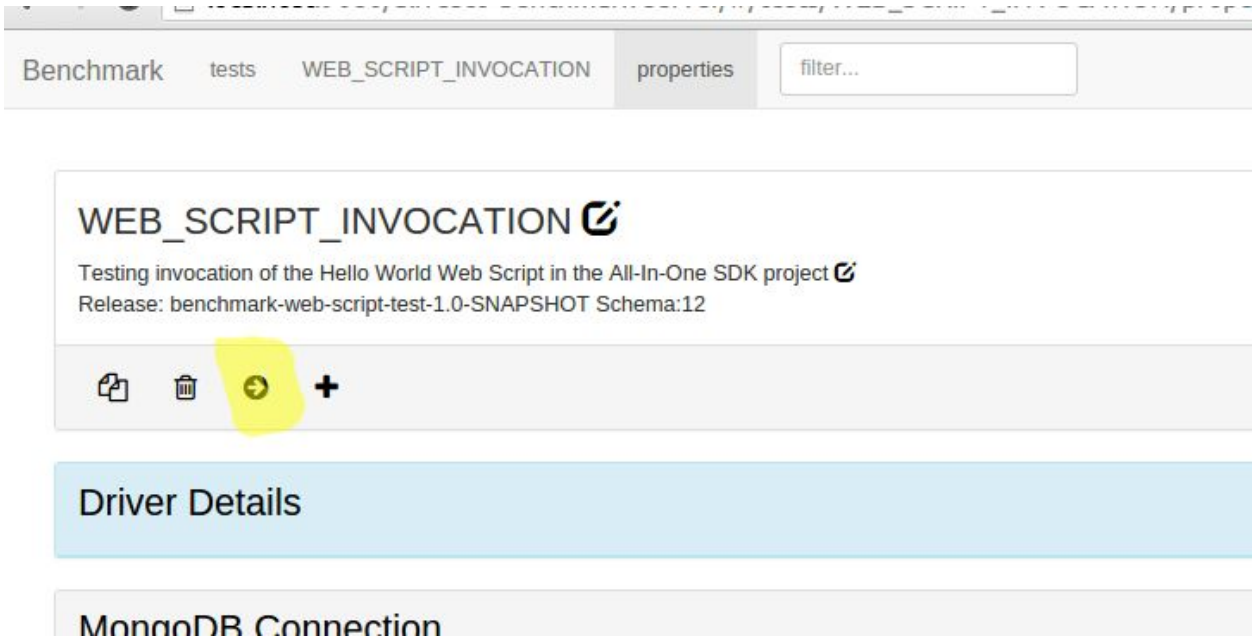
The only property that is required to be filled in is as usual the MongoDB server hostname. But let’s take a look at the rest of the property groups:



At the end of the property group list we can see our test specific groups, expanding the groups and we will see the properties we defined when developing the test:

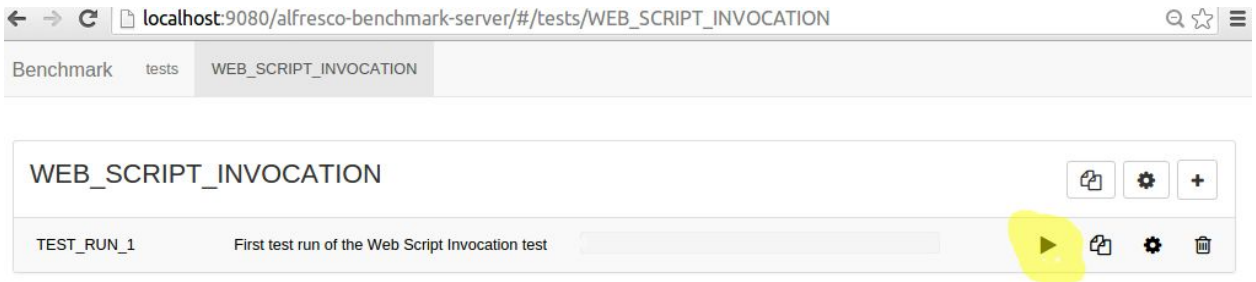


We will keep the default values for this test run. Click on the right arrow at the top to create a Test Run for this Test Definition:



This will display the following screen:

This is the first run with this new test definition so fill in the fields accordingly and then click **Ok:**

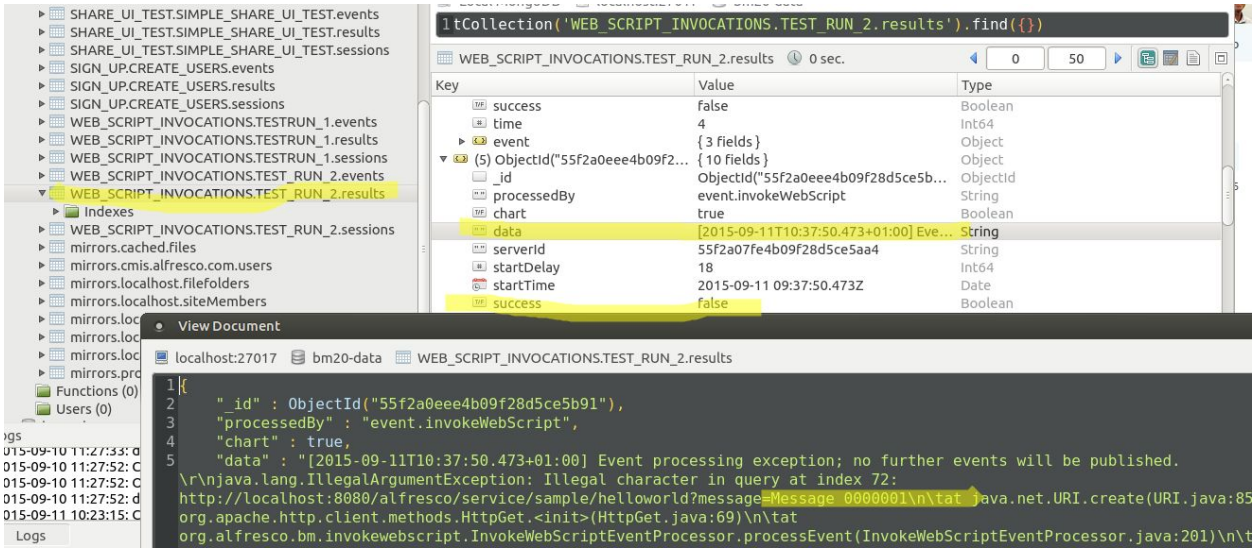


To start the load test click the “play” button.

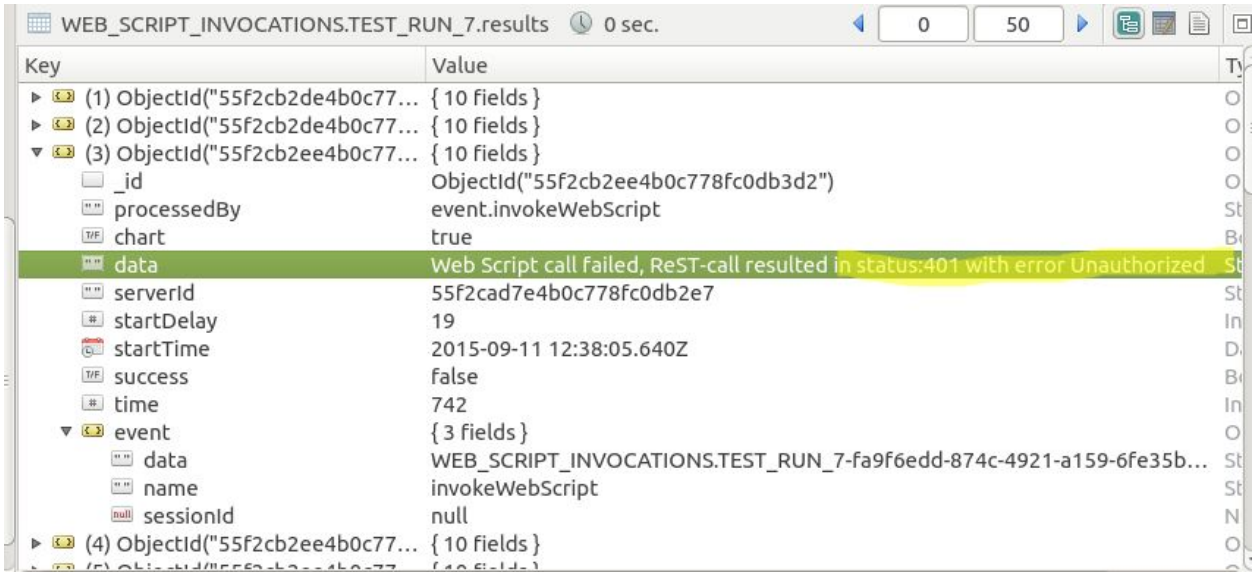
If you look at the Excel spreadsheet summary of the Test run something similar to this should be the result:

Name:	_INVOCATION.TEST_RUN_1						
Description:	the Web Script Invocation test						
Progress (%):	100						
State:	COMPLETED						
Started:	13-Sep-2015 14:45:24						
Finished:	13-Sep-2015 14:45:44						
Duration:	0:00:19.997						
Event Name	**Total Count**	**Success Count**	**Failure Count**	**Success Rate (%)**	**Min (ms)**	**Max (ms)**	**Arithmetic Mean (ms)**
invokeWebScript	200	200	0	100	7	1629	143
scheduleWebScriptInvocations	2	2	0	100	70	82	76

If you get errors then this would show up in the Event Result in MongoDB, such as in this example where I have not encoded the message parameter in the URL properly:



Or like in this example where I don't have any users in the Alfresco Repository:



So if something does not work always go in and check the Event Result.

Load Testing using Stand-alone Servers

In a more complex/real scenario you would install the Benchmark management and driver servers manually on each host on stand-alone Tomcat servers. This is useful when several load test implementations (WARs) should be supported from one Tomcat server.

References

https://wiki.alfresco.com/wiki/Benchmark_Framework_2.0
https://github.com/derekhulley/alfresco-benchmark
https://github.com/derekhulley/benchmark-cmis
https://github.com/AlfrescoBenchmark/alfresco-text-gen