

Docker CI/CD With Jenkins

Objective: This chapter explains how to use Jenkins and Docker to run continuous integration and continuous delivery

Preparation: Open up two instances of your favorite shell, and a browser window. Make absolute sure you are in the appropriate lab folder. Confirm you are in the Part B subfolder.

Outcome: Each student will replicate a highlight demo of some basic functionality

Data Files: Ask Instructor

There are several possible approaches to run Docker builds with Jenkins:

- Install Jenkins on your host machine, where Docker is also installed, and run Docker commands from your build, either using one of the several Jenkins Docker plugins, or by running Docker commands from a build step
- Install Jenkins on your host machine and have a Jenkins slave machine with Docker installed to run your Docker builds
- Run Jenkins on Docker and use the underlying Docker installed on the host to run Docker commands.

NOTE: Another option is running Jenkins on Docker and do a complete Docker installation inside the Jenkins Docker container. This technique is called Docker in Docker and it is usually a bad idea. There are

several discussions about the problems with this approach, like this one: <http://jpetazzo.github.io/2015/09/03/do-not-use-docker-in-docker-for-ci/>. A better approach is using Docker outside of Docker, as explained [here](#).

Step 1. Run Jenkins on Docker

In this example, we will run Jenkins on Docker and use the underlying Docker installed on the host to run Docker commands. This technique is known as Docker outside of Docker.

1. First, clone the project at

<https://github.com/Virtuant/jenkins-docker-demo>, however you should already have it.

2. Then, in the project folder, run:

```
docker-compose up
```

3. Wait for jenkins to start and then go to the browser and open

<http://localhost:8081>. Jenkins should be running.

The Jenkins installation on this lab comes pre-configured. To login use the username `jenkins` and the password `jenkins`.

Step 2. Running Integration Tests with

Docker and Jenkins

For this Continuous Integration demo, we will run a simple application that saves data on MongoDB. We will then run integration tests to check if the data was correctly saved on the database.

When running integration tests, you want to test your application in an environment as close to production as possible, so you can test interactions between the several components, services, databases, network communication, etc. Fortunately, docker can help you a lot with integration tests. There are several strategies to run integration tests, but in this application we are going to use the following:

- Start the services with a `docker-compose.yml` file created for testing purposes. This file won't have any volumes mapped, so when the test is over, no state will be saved. The test `docker-compose.yml` file won't publish any port on the host machine, so we can run simultaneous tests.
 - Run the application, using the services started with the `docker-compose.yml` test file.
 - Run Maven integration tests to check if the application execution produced the expected results. This will be done by checking what was saved on the MongoDB database.
 - Stop the services. No state will be stored, so next time you run the integration tests, you will have a clean environment.
1. Create a new job on the Jenkins Web UI by selecting "New Item," which you should see on the left-hand sided panel.

2. Select Freestyle project and give the project a name you will remember.
3. Click, “Ok” at the bottom on the list.

Jenkins

search

Jenkins > All >

Enter an item name

ci-test

» Required field

- Freestyle project**
This is the central feature of Jenkins. Jenkins will build your project, combining any SCM with any build system, and this can be even used for something other than software build.
- Maven project**
Build a maven project. Jenkins takes advantage of your POM files and drastically reduces the configuration.
- Multi-configuration project**
Suitable for projects that need a large number of different configurations, such as testing on multiple environments, platform-specific builds, etc.

OK

2. In Source Code Management, select Git and add the repository URL: <https://github.com/fabianenardon/mongo-docker-demo.git>

General

Source Code Management

Build Triggers

Build

Post-build Actions

Source Code Management

None

Git

Repositories

Repository URL

https://github.com/fabianenardon/mongo-docker-demo.git

Credentials

- none -

Add

Advanced...

Add Repository

Branches to build

Branch Specifier (blank for 'any')

*/master

Add Branch

Repository browser

(Auto)

Additional Behaviours

Add

Build Triggers

Save








Apply

from scripts)

Other projects are built

Note: If Git is not configured then please follow below steps and search for Git SCM plugin :



-  [Configure System](#)
Configure global settings and paths.
-  [Configure Global Security](#)
Secure Jenkins; define who is allowed to access/use the system.
-  [Configure Credentials](#)
Configure the credential providers and types
-  [Global Tool Configuration](#)
Configure tools, their locations and automatic installers.
-  [Reload Configuration from Disk](#)
Discard all the loaded data in memory and reload everything from file system. Useful when you modified config files dir
-  [Manage Plugins](#) ← **(updates available)**
Add, remove, disable or enable plugins that can extend the functionality of Jenkins.
-  [System Information](#)
Displays various environmental information to assist trouble-shooting.

Filter:

Updates
Available
Installed
Advanced

Install	Name	Version
.NET Development		
<input type="checkbox"/>	CCM This plug-in generates the trend report for CCM, an open source static code analysis program.	3.2
<input type="checkbox"/>	change-assembly-version-plugin	1.5.1
<input type="checkbox"/>	FxCop Runner	1.1
<input type="checkbox"/>	MSBuild Allows using MSBuild to build .NET projects	1.29
<input type="checkbox"/>	MSTest	0.23

3. In Build, select Add build step and select Execute shell

General Source Code Management **Build Triggers** Build Post-build Actions

Additional Behaviours **Add** ▼

Build Triggers

- ☐ Trigger builds remotely (e.g., from scripts) ?
- ☐ Build after other projects are built ?
- ☐ Build periodically ?
- ☐ GitHub hook trigger for GITScm polling ?
- ☐ Poll SCM ?

Build

Add build step ▼

- Execute Windows batch command
- Execute shell**
- Invoke top-level Maven targets
- Set build status to "pending" on GitHub commit

Save **Apply**

4. In the shell Command, add these instructions:

```
$ cd sample

# Generates the images
$ sudo /var/jenkins_home/tools/hudson.tasks.Maven_MavenInstallation/maven/bin/mvn clean install -Papp-docker-image

# Starts the mongo service. The -p option allows multiple builds to run at the same time,
# since we can start multiple instances of the containers
```

```
$ sudo docker-compose -p app-$BUILD_NUMBER --file src/test/re  
sources/docker-compose.yml up -d mongo
```

```
# Waits for containers to start
```

```
sleep 30
```

```
# Run the application
```

```
$ sudo docker-compose -p app-$BUILD_NUMBER --file src/test/re  
sources/docker-compose.yml \
```

```
    run mongo-docker-demo \
```

```
        java -jar /maven/jar/mongo-docker-demo-1.0-SNAPSHOT-jar-  
with-dependencies.jar mongo
```

```
# Run the integration tests
```

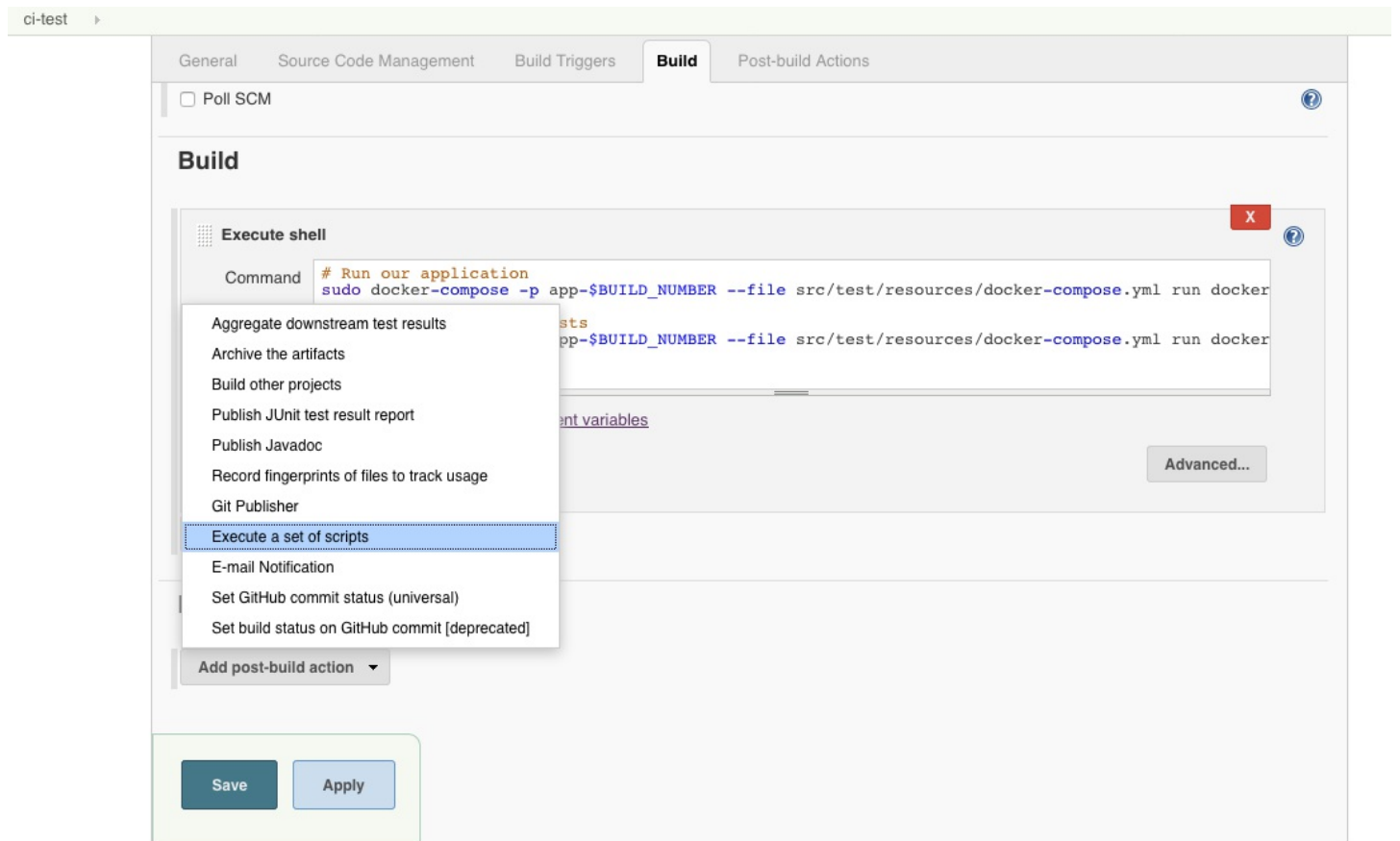
```
$ sudo docker-compose -p app-$BUILD_NUMBER --file src/test/re  
sources/docker-compose.yml \
```

```
    run mongo-docker-demo-tests \
```

```
        mvn -f /maven/code/pom.xml -Dmaven.repo.local=/m2/reposi  
tory \
```

```
        -Pintegration-test verify
```

5 . Click on “Add post-build action” and select “Execute a set of scripts”



Note: If you are not able to find “Execute a set of scripts” option then look for option shown in below screenshots:

Post-build Actions

Add post-build action ▾

- Aggregate downstream test results
- Archive the artifacts
- Build other projects
- Publish JUnit test result report
- Publish Javadoc
- Record fingerprints of files to track usage
- Git Publisher
- Pretested Integration publisher to push to integration branch
- E-mail Notification
- Execute Scripts**
- Set GitHub commit status (universal)
- Set build status on GitHub commit [deprecated]

Add post-build action ▾

Save

Apply

Post-build Actions

Execute Scripts

Add generic script file

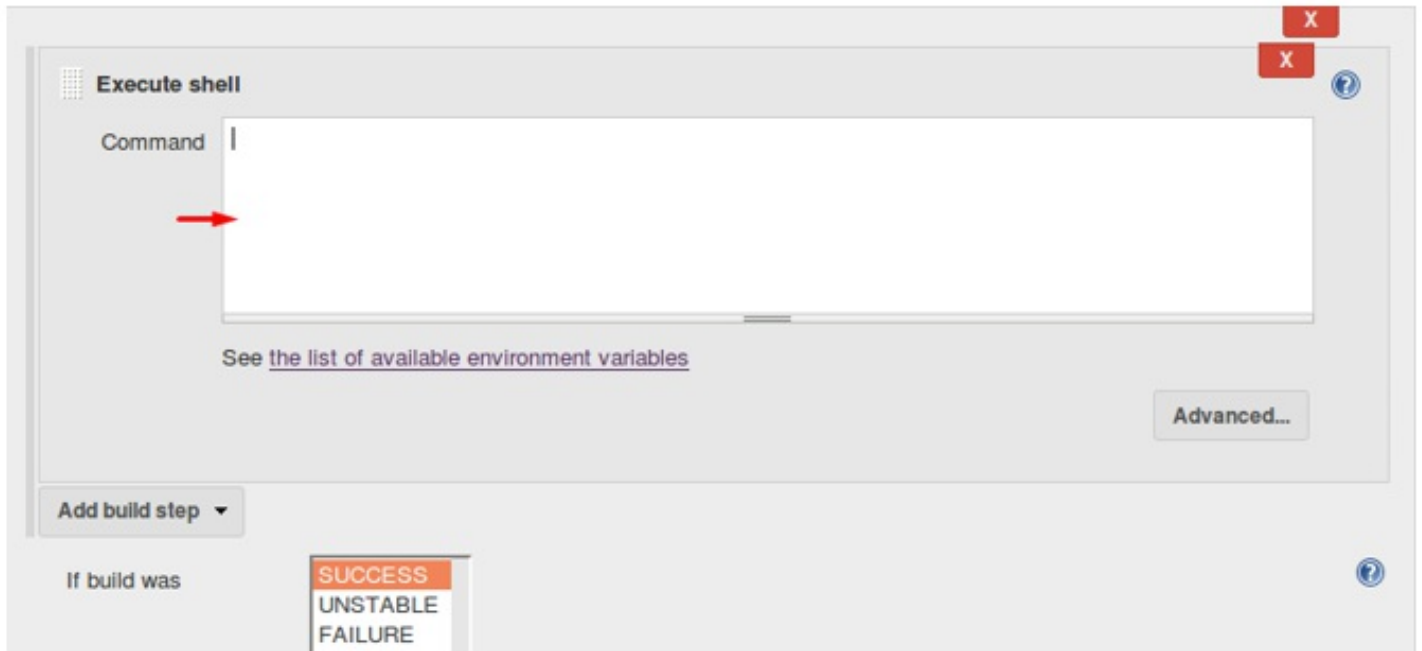
Add Groovy script file

Add Groovy script

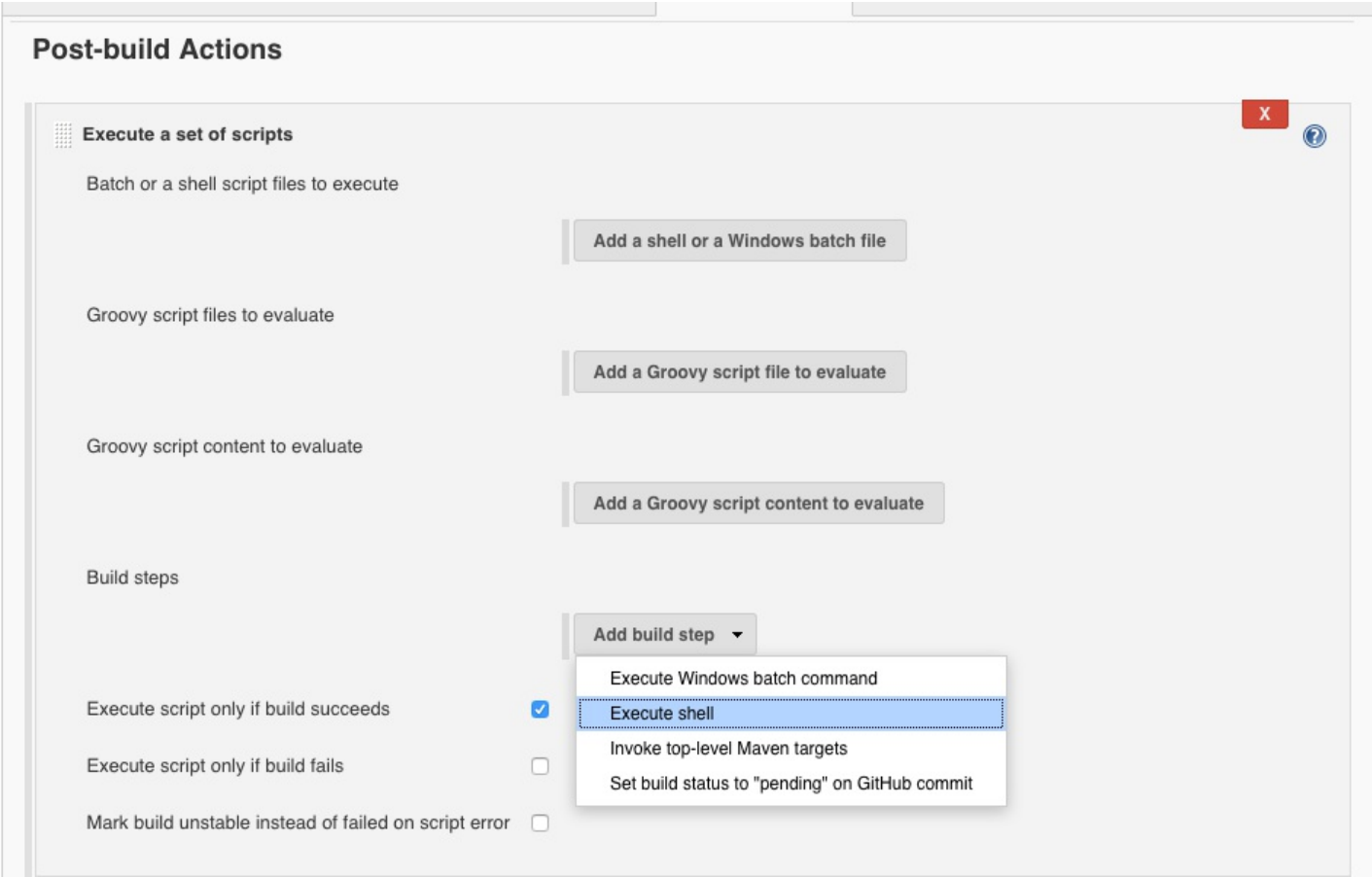
Add post build step

☐ Mark build unstable

Add post-build action ▾



6 . Under “Post-build Actions”, click on “Add build step” dropdown menu and select Execute shell



7 . In the Command box, add:

```
$ cd sample  
$ sudo docker-compose -p app-$BUILD_NUMBER --file src/test/resources/docker-compose.yml down
```

8 . Uncheck the **Execute script only if build succeeds** and **Execute script only if build fails** options, so this script will run always when the build ends. This way, we make sure to always stop the services.

NOTE:

- The **-p app-\$BUILD_NUMBER** option allows multiple builds to run at the same time, since we can start multiple instances of the containers. We are using Jenkins \$BUILD_NUMBER variable to isolate the containers. This way, each set of services will run on its own network.
- We are running the commands with sudo because we are actually running the Docker socket on the host. Jenkins runs with the jenkins user and we added the jenkins user to the sudoers list in our image. Obviously, this can have security consequences in a production environment, since one could create a build that would access root level services on the host. You can avoid this by configuring the jenkins user on the host, so it will have access to the Docker socket and then run the commands without sudo.

9 . Save the build and then click on **Build now** to run it. You should see a

progress bar when the build is running. You can click on the progress bar to see the build console output.

The screenshot shows the Jenkins web interface for a project named 'ci-test'. The top navigation bar includes the Jenkins logo and a red tab with the number '1'. Below the navigation bar, there are several links: 'Back to Dashboard', 'Status', 'Changes', 'Workspace', 'Build Now', 'Delete Project', and 'Configure'. The main heading is 'Project ci-test'. To the right of the heading, there are links for 'Workspace' and 'Recent Changes'. Below the heading, there is a 'Build History' section with a search bar and a 'trend' link. The build history table shows three builds: #3 (blue icon, Apr 8, 2017 12:53 AM), #2 (blue icon, Apr 8, 2017 12:54 AM), and #1 (red icon, Apr 8, 2017 12:54 AM). A red arrow points to the progress bar of build #3, which is highlighted with a tooltip showing 'Started 18 sec ago' and 'Estimated remaining time: 2 min 22 sec'. To the right of the build history, there is a 'Permalinks' section with a list of links: 'Last build (#2), 1 min 55 sec ago', 'Last stable build (#2), 1 min 55 sec ago', 'Last successful build (#2), 1 min 55 sec ago', 'Last failed build (#1), 6 min 33 sec ago', 'Last unsuccessful build (#1), 6 min 33 sec ago', and 'Last completed build (#2), 1 min 55 sec ago'. At the bottom of the build history section, there are links for 'RSS for all' and 'RSS for failures'.

Jenkins

Jenkins > ci-test >

Back to Dashboard

Status

Changes

Workspace

Build Now

Delete Project

Configure

Project ci-test

Workspace

Recent Changes

Build History

find x

trend

Build	Time	Status
#3	Apr 8, 2017 12:53 AM	Running
#2	Apr 8, 2017 12:54 AM	Completed
#1	Apr 8, 2017 12:54 AM	Failed

Started 18 sec ago
Estimated remaining time: 2 min 22 sec

RSS for all RSS for failures

Permalinks

- [Last build \(#2\), 1 min 55 sec ago](#)
- [Last stable build \(#2\), 1 min 55 sec ago](#)
- [Last successful build \(#2\), 1 min 55 sec ago](#)
- [Last failed build \(#1\), 6 min 33 sec ago](#)
- [Last unsuccessful build \(#1\), 6 min 33 sec ago](#)
- [Last completed build \(#2\), 1 min 55 sec ago](#)

10 . If the build is successful, you should see this on the build console output:

Tests run: 1, Failures: 0, Errors: 0, Skipped: 0

```
[WARNING] File encoding has not been set, using platform encoding UTF-8, i.e. build is platform dependent! The file
encoding for reports output files should be provided by the POM property ${project.reporting.outputEncoding}.
[INFO]
[INFO] --- maven-failsafe-plugin:2.19.1:verify (verify) @ mongo-docker-demo ---
[WARNING] File encoding has not been set, using platform encoding UTF-8, i.e. build is platform dependent! The file
encoding for reports output files should be provided by the POM property ${project.reporting.outputEncoding}.
[INFO] -----
[INFO] BUILD SUCCESS
[INFO] -----
[INFO] Total time: 7.998 s
[INFO] Finished at: 2017-04-08T00:55:31+00:00
[INFO] Final Memory: 18M/170M
[INFO] -----
[PostBuildScript] - Execution post build scripts.
[ci-test] $ /bin/sh -xe /tmp/hudson8483450880790053105.sh
+ cd sample
+ sudo docker-compose -p app-3 --file src/test/resources/docker-compose.yml down
Stopping app3_mongo_1 ...
[1A]2K
Stopping app3_mongo_1 ... done
[1BRemoving app3_mongo-docker-demo-tests_run_1 ...
Removing app3_mongo-docker-demo_run_1 ...
Removing app3_mongo_1 ...
[2A]2K
Removing app3_mongo-docker-demo_run_1 ... done
[2B[1A]2K
Removing app3_mongo_1 ... done
[1B[3A]2K
Removing app3_mongo-docker-demo-tests_run_1 ... done
[3BRemoving network app3_default
Finished: SUCCESS
```

Running and Debugging Integration Tests Outside Jenkins

1. When creating integration tests, it is useful to be able to run and debug them outside Jenkins. In order to do that, you can simply run the same commands you ran in the Jenkins build:

```
# Generates the images
```

```
mvn clean install -Papp-docker-image
```

```
# Starts mongo service
```

```
docker-compose --file src/test/resources/docker-compose.yml u
```

```
p -d mongo
```

```
# Waits for services do start
```

```
sleep 30
```

```
# Run our application
```

```
docker-compose --file src/test/resources/docker-compose.yml \  
    run mongo-docker-demo \  
    java -jar /maven/jar/mongo-docker-demo-1.0-SNA  
PSHOT-jar-with-dependencies.jar mongo
```

```
# Run our integration tests
```

```
docker-compose --file src/test/resources/docker-compose.yml \  
    run mongo-docker-demo-tests mvn -f /maven/code  
/pom.xml \  
    -Dmaven.repo.local=/m2/repository -Pintegratio  
n-test verify
```

```
# Stop all the services
```

```
docker-compose --file src/test/resources/docker-compose.yml d  
own
```

2. If you wanted to debug your integration tests, you would run the tests with this command:

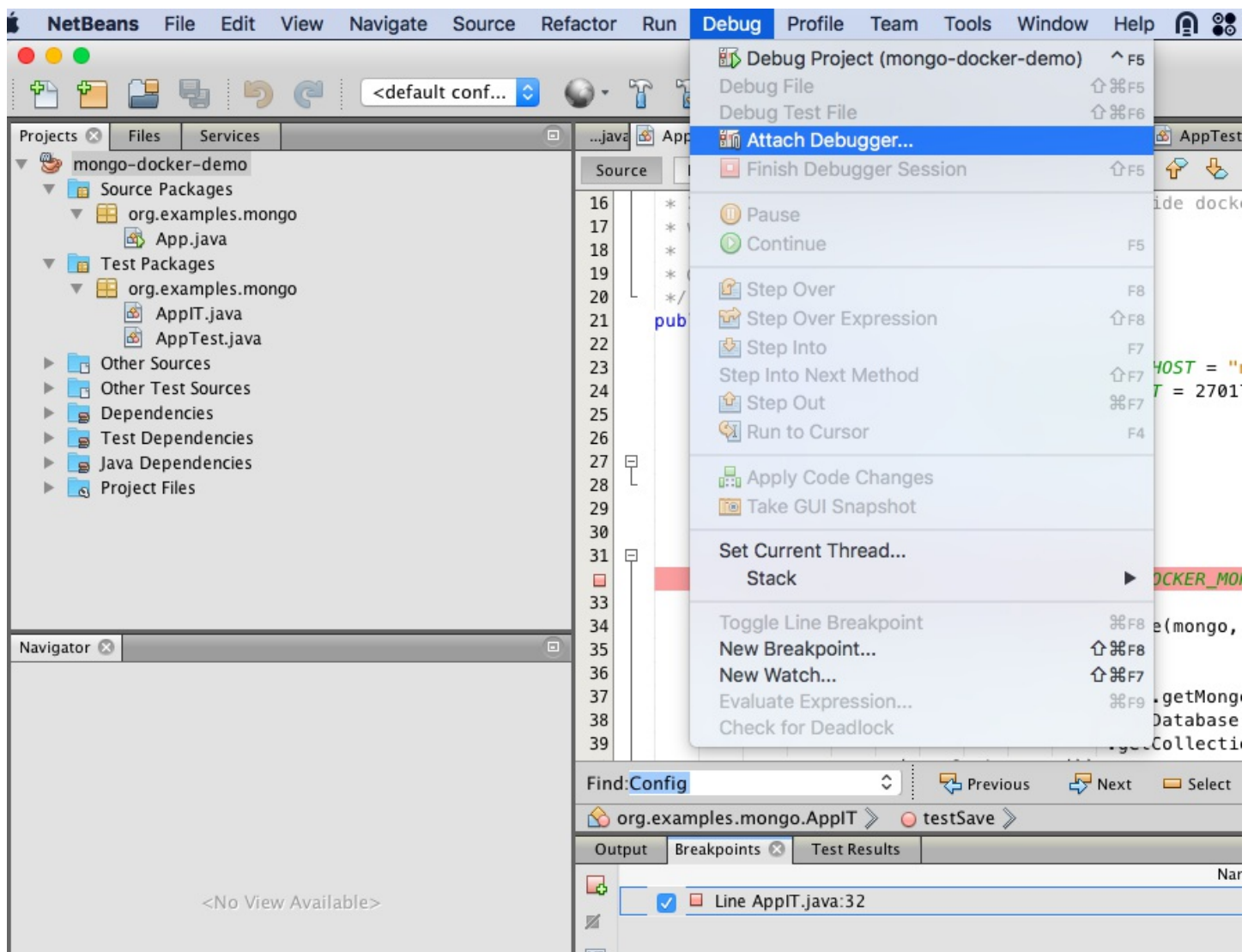
```
# Run integration tests in debug mode
```

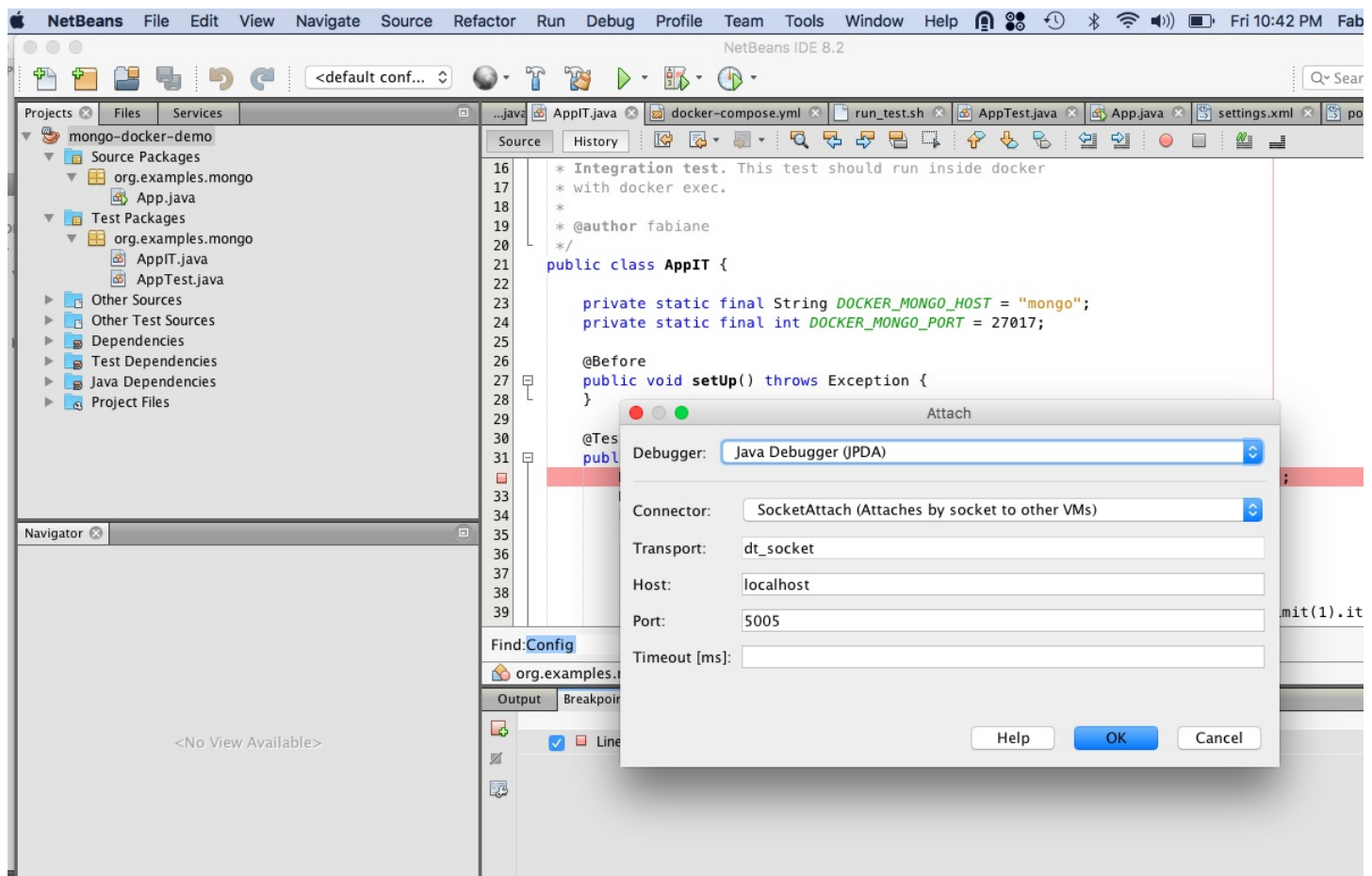
```
docker run -v ~/.m2/repository:/m2/repository \  
    -p 5005:5005 --link mongo:mongo \  
    mongo-docker-demo-tests mvn -f /maven/code  
/pom.xml -Dmaven.repo.local=/m2/repository -Pintegration-test verify
```

```
--net resources_default mongo-docker-demo-tests \  
mvn -f /maven/code/pom.xml \  
-Dmaven.repo.local=/m2/repository \  
-Pintegration-test verify -Dmaven.failsafe.debug
```

3. This will make your test wait for a connection on port 5005 for debugging. You can then attach your IDE to this port and debug.

Here is how this is done on Netbeans:





Conclusion:

Continuous Delivery strategies depend greatly on the application architecture. With a dockerized application like the one in our demo, the continuous delivery strategy could be to publish a new version of the application image if the tests passed. This way, next time the application runs on production, the new image will be downloaded and automatically deployed. You can publish images with Jenkins just like you invoked all the other docker commands in the build.