Jenkins
Starter Guide Ebook
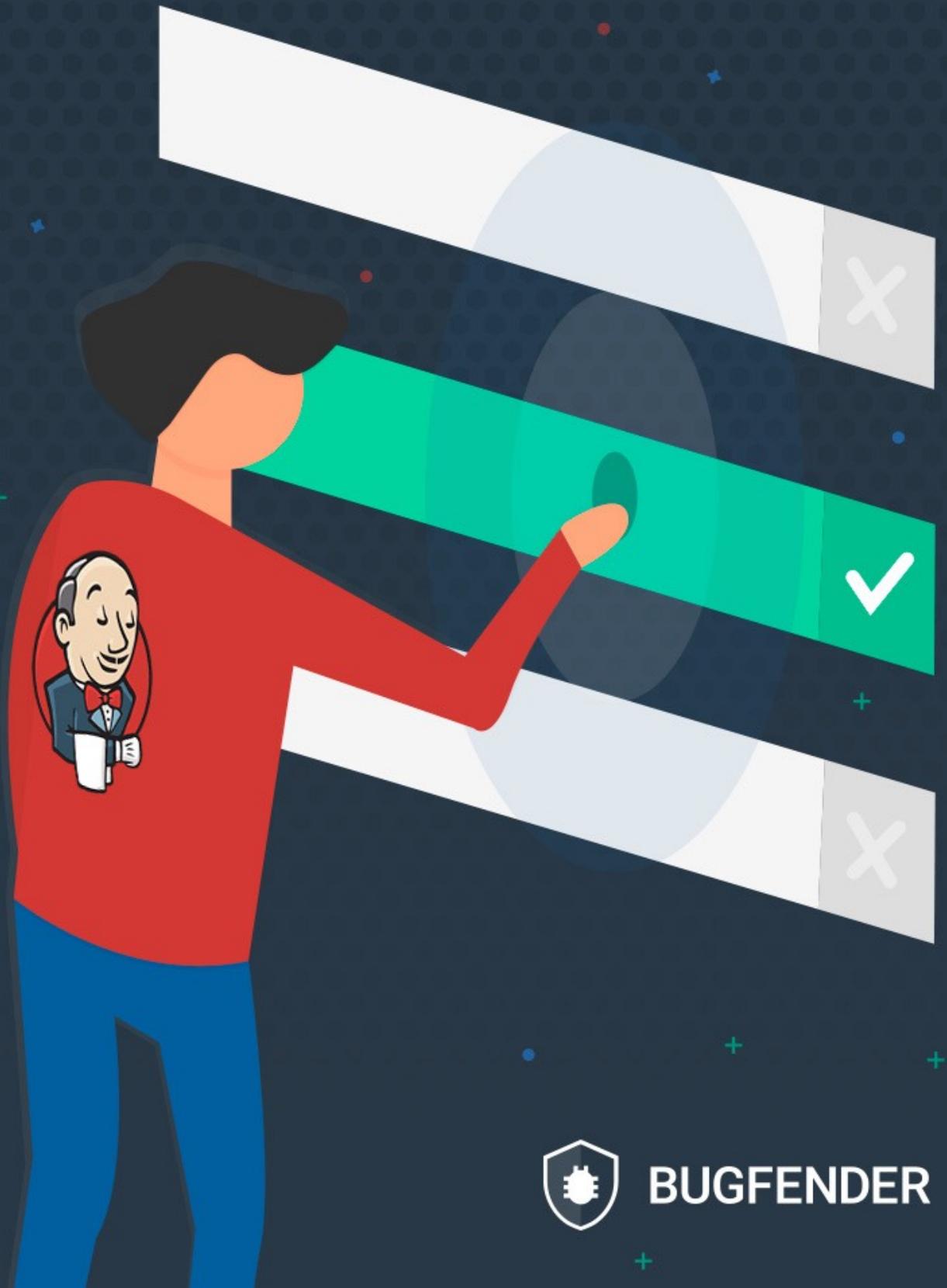
BUGFENDER

Table of Contents

Forward

Why We Use Jenkins at Bugfender

We designed Bugfender to meet our own internal need for remote logging capabilities for mobile app development at Mobile Jazz. We launched it as a product in its own right in 2015.

We took a common mobile developer headache and made it a little simpler. In the beginning, several of our developers realized they needed a remote way to store users' logs, so they set up the infrastructure to make it happen.

As this side project has become a business of its own right, much of our progress can be credited to all the work that has gone into **automating and maintaining cloud-based development, testing, and deployment environments**.

About 4 years ago, we realized that although we would write good iOS unit tests, we did not execute them regularly enough to make sure they kept working.

Writing tests without using them is a complete waste of time, so we decided to go one step further: using Jenkins to automate the execution of the tests every time someone makes a change to the code.

In this way, Jenkins has allowed us to cut excess by reducing the time spent manually running tests. At Bugfender, we like to home-cook solutions to automate as much as possible, so this open source software worked for us.

BUGFENDER

Jenkins makes sure our code compiles, so we save hundreds of hours of manpower because we don't have to work through the tiny issues in compilation and shipping to a production-ready state. Every commit that is green is deployable to production. Boom, we're done.

At its core, CI/CD is all about letting machines do what they do best so humans can do what brings the most value to your company.

Of course, one of the main obstacles is **belief** in the value of CD/CI at every company level. As we will explain in the first chapter of in this ebook, there are complications to maintaining a CI/CD environment, and many developers find themselves trying to convince upper management why it's worth it.

Fortunately, the hub of enterprise for Jenkins and DevOps, CloudBees, has put a number on the savings that businesses can expect from integrating continuous delivery in their devops teams. **CI/CD is all about saving your company money:**

> *A CloudBees assessment of more than 100 enterprises revealed that continuous delivery enables an average efficiency gain of 66 hours per developer per year. For a 100-person team, this efficiency equates to 6,600 more hours to invest in innovation; an estimated annual value of $350,000[1].*

Currently, though it seems CI/CD is a popular talking point, many teams haven't made the leap yet or neglect the upkeep needed to keep their testing environment operative.

We created this ebook to help your company fast-track the implementation of Jenkins, so you can start saving money + start preventing a common developer headache.

BUGFENDER

What is Jenkins and Why Should You Be Using It?

A common problem for many devops teams is a fragmented workflow.

Inefficiency can be **infuriating**.

You know how it goes: individuals on the team tend to work independently. Coding solo, engineers regularly create large segments of code outside of version control. Once a developer is "done," they add their work into the basecode. Then another team manually runs tests to verify the build.

For years, many teams have found this division of labor annoying and problematic.

When multiple developers separately commit large changes to version control, they create complex bugs, multiply time-intensive fixes, and increase the time it takes to do more manual testing. Everything slows down.

These inefficiencies clog up your build cycle with tedious debugging, slow down the time to production, and ultimately *undercut your company's profits.*

What is Continuous Integration?

Continuous integration (CI) is a process aimed at cutting out build cycle inefficiencies by allowing developers to compile the team's code from a shared version control repository. CI also allows you to automate testing so you can set up the system to automatically run unit tests or integration tests for example.

CI automatically monitors the commits that each engineer makes. This streamlines the build and verification of code so that testing is not so high-stakes. It's recognized as a best practice in the software development community.

CI is run on a shared server that increases visibility, so all the engineers on a project are aware of changes in the base code day in and day out. In addition, you can configure the server to alert developers when they submit failing code so that they can fix any errors they introduce.

Using CI automation allows you to **shorten development release cycles** and **improve product quality**. And it's all customizable to your project and your needs.

CI/CD servers, including Jenkins, allow your team to set up the tests that you need to run.

BUGFENDER

There are various levels of testing automation that you can implement:

The most basic test is whether or not the code actually compiles. Code can also be "linted" or checked for style. Your team can write more complex tests to cover other bases as well, including unit, integration, stress, regression testing, etc.
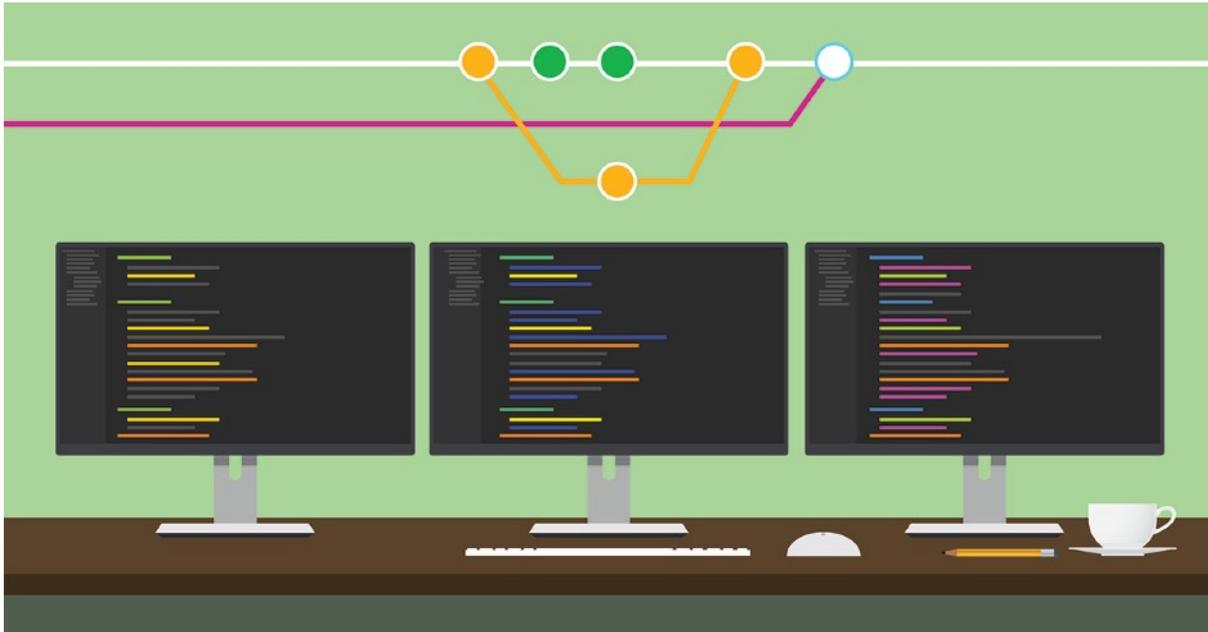
How Does Continuous Integration Differ from Continuous Delivery?

A CI service compiles and tests a full application (possibly by running the application in an emulator). Continuous delivery also pushes this compiled application to a repository, for example, for alpha testers to use and provide early feedback.

CD builds are automatically deployed through to the production environment and can also be used for broader beta testing.

CD is aimed at lean-logistics: automating the process from adding new code through to acceptance testing. CD automates all the steps so that your build is ready to be deployed.

→ Continuous delivery: Besides building the application and running tests on it, the application is also "delivered," which often means putting it on a server for someone to perform manual tests on it, or sent by email to a test group of users (eg. a mobile app sent by email).

→ For production builds, delivery means deploying the application for the end users.

This allows for faster, smaller deployments of your product which reduce deployment risk. Regular, smaller, leaner deliveries are less risky that huge ones that only occur once or twice a year.

How Does CI/CD Benefit a Devops Team?

A lot of people talk about CI because it is a best practice to have all code verified automatically on every commit, regularly run tests, and continuously deploy–for the reasons mentioned above. The process checks that the contributions from each developer work well together. Detecting these issues early makes it easier and faster to fix the bugs.

As you can probably guess, **implementing CI/CD will cause a cultural shift in your team.** You will have to become more agile and integrated.

As we mentioned in our intro, CI was created to allow your team to stop wasting human time trying to manually resolve conflicting segments of new code, trigger builds, or run tests.

Instead, it encourages adding small, incremental changes in code so you don't end up with huge, complex bugs to fix. This speeds up the build cycle, potentially streamlining deployment and production.

And it's helpful to remember—production is where your business actually makes money.

Also, interestingly, for a multi-developer remote team, CI is extremely helpful to implement because it brings you all together and continuously combines your work so you're all on the same page regardless of geographic location.

Jenkins helps structure the build cycle of a remote team.

How Jenkins Helps the Bugfender Team:

➔ Running Jenkins allows us to continuously check that the code compiles and run a set of basic tests that we have programmed to verify the base code after every commit.

➔ We automatically deploy the merged code to an integration environment, which can be used for manual tests (this bit is called "continuous delivery").

➔ We also use it to generate and deploy production builds. Our deployments are complex because we are not updating a single machine: We're updating a whole cluster of machines. We have to be sure to perform updates in a way that does not disrupt the service. Deployment in such an environment could be prone to errors if done by a human.

Intro to Jenkins

Jenkins is an open source implementation of a CI server written in Java that can be used as a self-hosted option automating the build cycle for any project. It works with any programming language and for multiple platforms including Windows, Linux and macOS. According to the Jenkins website:

> Jenkins, originally founded in 2006 as "Hudson", is one of the leading automation servers available. Using an extensible, plugin-based architecture developers have created hundreds of plugins to adapt Jenkins to a multitude of build, test, and deployment automation workloads. In 2015, Jenkins surpassed 100,000 known installations making it the most widely deployed automation server.

One of the main benefits of Jenkins is that it is a well-known tool with lots of community support, there are many plugins available (including well-known names like Slack, GitHub, Docker, Build Pipeline + more), and the project is well-maintained by a large community of developers.

There are a lot of conversations going on about CI/CD and the Jenkins project. But even as late as 2017, a surprising number teams don't utilize or

maintain their systems because **running a CI environment is costly, either in time or money, often both**.

So why even bother with them? Let's look at that question.

Pros vs. Cons of Jenkins

Jenkins is an old tool with an unfriendly user interface.*

(*The good news is that the Jenkins project just released Blue Ocean, a continuous delivery software aimed at significantly improving the UI. True to form, it is 100% open source.)

As your team considers options for continuous integration, it is helpful to note that Jenkins must be run on a server (cost), so it often needs the attention of someone with system administration skills (time). You can't just set it up and then expect it to run itself–the system requires frequent updates and maintenance.

Yet, Jenkins is open source and one of the best and most widely used free tools out there for implementing CI/CD for your devops team.

The main barrier to entry for most teams is the initial setup, procrastination, or failed previous attempts to set it up.

People tend to know it's a best practice, but many teams neglect it for more urgent coding work. Perhaps someone on your team tried to implement Jenkins at some point, but did not successfully maintain it. Maybe the wasted effort gave your boss a bad impression about it.

As it is, the reasons people do not implement a CI server are usually very practical.

One main reason: CI systems regularly break. If a setting in the project changes, often it is necessary to readjust the configuration of the CI system. If the CI system is not perceived as highly valuable by the team, they tend to leave it aside, broken, so it stops delivering value.

Yet another reason for not using CI is that you need to write tests. Writing tests is something most developers want to do (i.e. a best practice), but they often don't find the time to do. Understandably, coding the actual software is usually a higher priority for business than more administrative tasks.

Also, tests break, meaning when the functionality under a test changes, it needs to be updated. If they're not updated, they stop delivering value, as in the case above. You have to prioritize maintaining the infrastructure yourself or it will not work.

In summary, it takes time to set up and a decent amount of ongoing work to keep it updated. But your team can adapt to streamline the maintenance of the system and tests to increase efficiency.

Of course, there are hosted SaaS alternatives to Jenkins, which could be beneficial if you're willing to pay a bit extra for someone else to maintain the software. Businesses tend to choose this option when they need a superior UI than what Jenkins offers. But a major benefit of self-hosting is that you have more control over your own data security.

Implementing CI requires a cultural shift, especially from the management. They have to allow time for this "unproductive stuff" to be done, while some of other day to day tasks go on hold.

BUGFENDER

Still, the brief sacrifice of time translates into long-term benefits for the whole company ($$$). With Jenkins, your code is easier to maintain and fewer bugs sneak into production. Your team becomes more integrated. Builds take less time. Your business can ship faster and keep up with the changing needs of your customers.

All of this will require a mindset shift:

CI is not an expense but an investment. And the ROI for implementation can be counted in time saved, errors avoided, and higher quality products delivered more easily to your clients.

BUGFENDER

How to Install Jenkins in Ubuntu

If you're getting started with Jenkins, there are many things to consider before you hit download.

You have to choose where to run Jenkins: on hardware that you maintain yourself or through a hosting service. You should work with your team to decide on the best option for your needs.

As a fully remote lifestyle business, we chose to use Amazon Web Services on a t2.nano machine type with Ubuntu 16.04, but this configuration can be adapted depending on the use case.

Selecting the Right Machine

Make sure you choose a machine with at least 1GB of RAM. If you don't have enough internal memory, Jenkins won't run. For decent performance, 2 to 4GB is ideal.

For Android, we would recommend a t2.medium machine at least. For iOS, you need a Mac, so we would recommend a Mac Mini. Many people find an older Macbook around the office and use it to run Jenkins, though we found that ours got overheated so we had to upgrade.

Choosing an Operating System

Jenkins runs on nearly any operating system you can imagine, as long as it can run Java or Docker. For this post I will be using Ubuntu, but you can use whatever operating system suits you best because the steps are very similar.

How to Install Jenkins

Step 1: Install Jenkins Using your Package Manager

On Ubuntu you can do this with:

```
wget -q -O - https://pkg.jenkins.io/debian-stable/jenkins.io.key |

sudo apt-key add -

echo "deb http://pkg.jenkins.io/debian-stable binary/" | sudo
tee /etc/apt/sources.list.d/jenkins.list

sudo apt-get update && sudo apt-get -y install jenkins
```

Note: You have other installation options including two different release lines. See https://jenkins.io/download/ to learn the difference between weekly release and LTS (long-term support) options.

Once installed, Jenkins will be running on port 8080. If you are accessing this server remotely, make sure to open the port 8080 in your firewall.

BUGFENDER

Step 2: Open Your Browser

In your browser, go to http://your-ci-server-name.com:8080. If your machine is running Jenkins, the window should display the setup wizard.

Step 3: Unlocking Jenkins
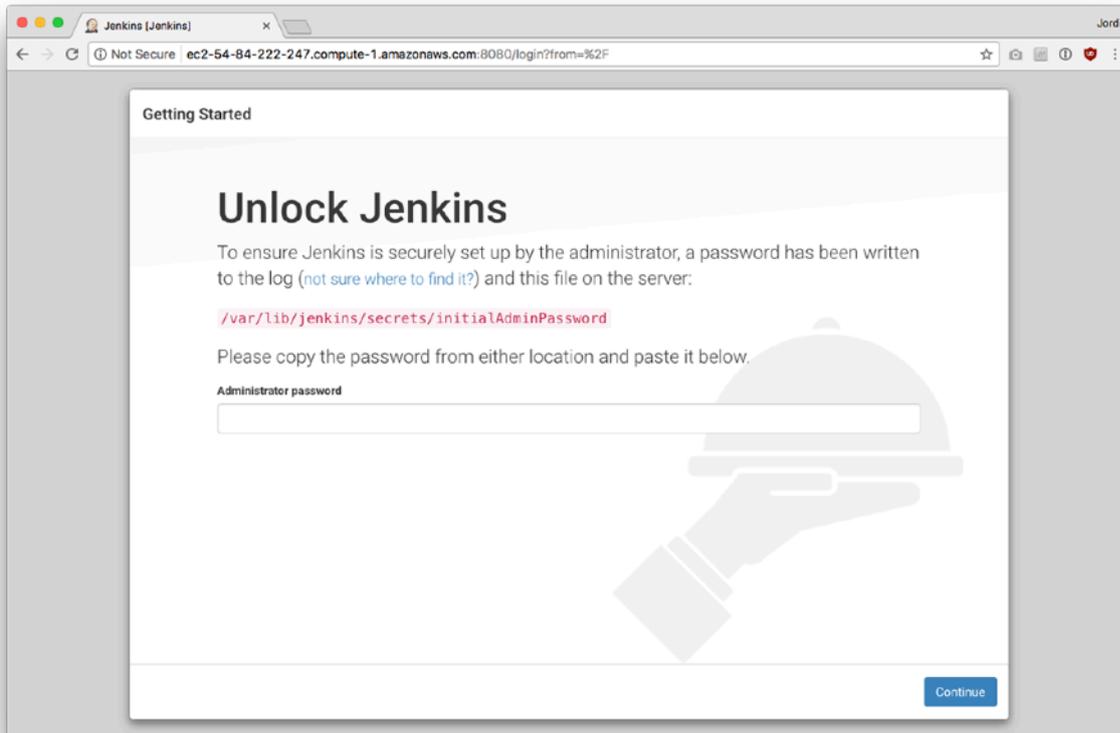
In order to ensure that you're the real administrator of the machine, the password has been written to a specified log file. In a fresh terminal, type in `sudo cat` and then paste in the pathway provided on this page in order to get a temporary password. Then you can enter the password in the **Administrator password** field in your browser.

This step is a security measure in case your Jenkins download is publicly accessible from the internet or your company's network.
(For information about more security measures for Jenkins, see "Recommended detour: add SSL to protect your Jenkins installation" below.)

BUGFENDER

Step 4: Customizing Jenkins Plugins
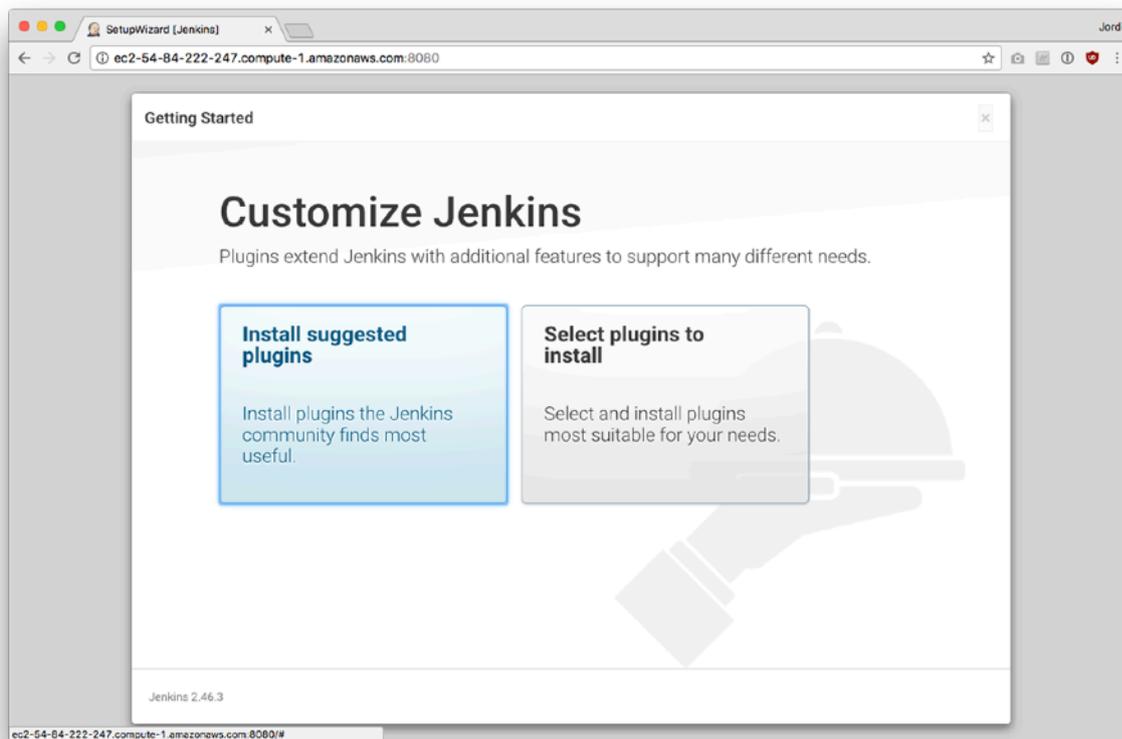
You can go ahead and click the **Install the suggested plugins box**. This will activate the download of a handful of useful plugins, including Git and some Pipeline plugins. You can always download other plugins as necessary to supplement these pre-selected options.

BUGFENDER

Step 5: Create Admin Login
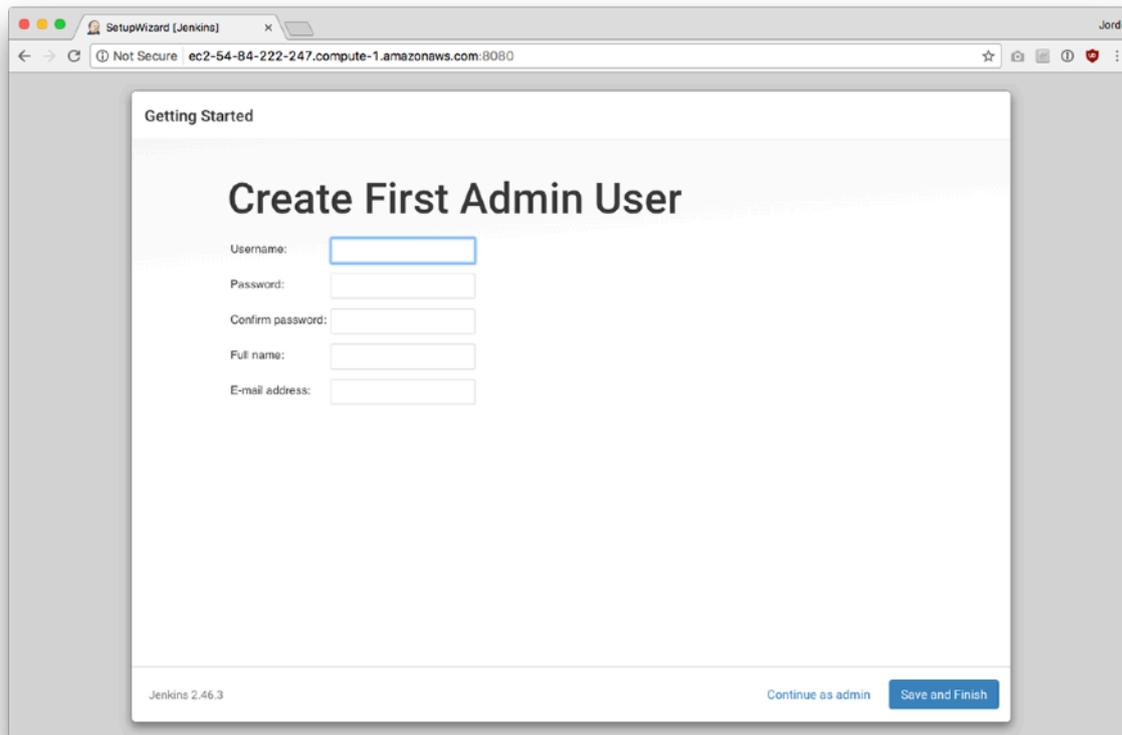
You can create your user account by adding a username, entering a new password, and including a full name and email address. Once you're in Jenkins, you can choose to enable new users which will allow colleagues to create additional logins as needed.

And that's it! You've installed Jenkins.

BUGFENDER
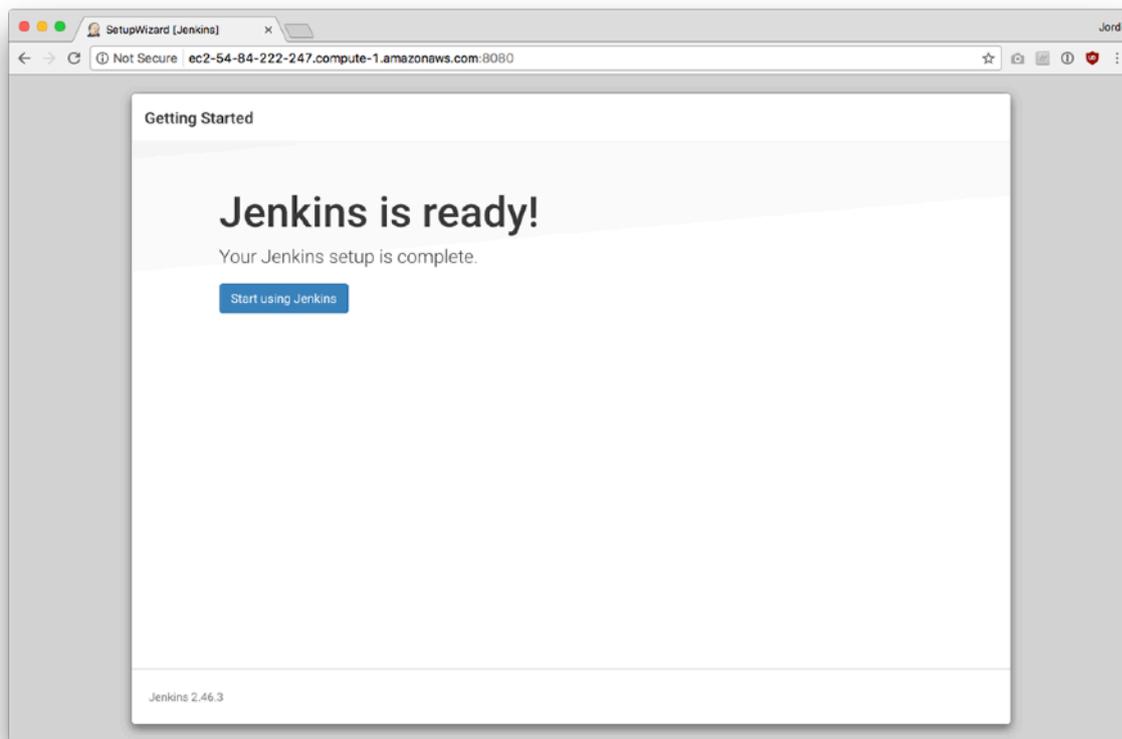
Recommended Detour: Add SSL to Protect Your Jenkins Installation

Since Jenkins has access to your application's source code, we recommend you use encryption. The essence of your product–the code–is likely sensitive and should be kept private. If you are running Jenkins through the internet or on a local network shared with other people, it might be good to protect communications with SSL.

Since the launch of Let's Encrypt, it's fairly easy and completely free to get these certificates, so we highly recommend it. Don't be lazy!

Step 1: If your server is behind a firewall, first make sure you are accepting inbound traffic on ports 80 and 443, and remove access to port 8080.

In order to get SSL working, you'll first need a DNS name for your server. In this example, we are calling it your-ci-server-name.com. Make sure to edit the

BUGFENDER

configuration files we will be listing here to match the domain name you choose.

Step 2: Then install nginx (`sudo apt-get install nginx`) and edit your /etc/nginx/sites-enabled/default file to look like this:

```
upstream app_server {
    server 127.0.0.1:8080 fail_timeout=0;
}

server {
    listen 80 default_server;
    listen [::]:80 default_server;
    server_name _;
    location ^~ /.well-known/acme-challenge/ {
        alias /usr/share/nginx/html/.well-known/acme-challenge/;
    }
    location / {
        return 301 https://$host$request_uri;
    }
}
```

Once it is installed, ask nginx to load your changes: `service nginx reload`

This will let the Let's Encrypt validation service talk to your server and verify that you own it.

Step 3: Now you can install the Let's Encrypt agent and make an SSL certificate for your server:

```
sudo apt-get install letsencrypt
Letsencrypt
```

Follow the steps of the wizard and you will end up with your SSL certificates in a directory like /etc/letsencrypt/live/your-ci-server-name.com/.

BUGFENDER

Now edit again /etc/nginx/sites-enabled/default to use those certificates, add the following lines:

```
server {
    listen        443;
    server_name  your-ci-server-name.com;
    ssl     on;
    ssl_certificate    /etc/letsencrypt/live/your-ci-server-
name.com/fullchain.pem;
    ssl_certificate_key    /etc/letsencrypt/live/your-ci-server-
name.com/privkey.pem;

    location / {
        proxy_set_header X-Forwarded-For
$proxy_add_x_forwarded_for;
        proxy_set_header X-Forwarded-Proto https;
        proxy_set_header Host $http_host;
        proxy_redirect off;

        if (!-f $request_filename) {
            proxy_pass http://app_server;
            break;
        }
    }
}
```
Then again ask nginx to load your changes: `service nginx reload`

Step 4: Now you should have SSL, try opening https://your-ci-server-name.com on your browser. If you followed these steps correctly you will looking at your brand new encrypted Jenkins installation.

Now that Jenkins is fully installed, keep reading to learn how to add your first Android and iOS jobs.

How to Add your First Android Job to Jenkins

Now you're ready to go. In this demo, we'll use an example application using Bugfender SDK in Android.

You'll obviously want use your own built Android application or library. The steps for adding your own Android project to Jenkins will be the same.

Building iOS apps? Check out the next section for iOS builds.

First off, you'll need to prepare the machine to compile and run Android applications. If you haven't already installed Jenkins, you can follow the steps in the previous post to install Jenkins in your server.

BUGFENDER

Configuring Android App Builds in Jenkins

Step 1: Install Java JDK

```
sudo apt-get install java-8-openjdk
```

Step 2: Install Android SDK

Go to https://developer.android.com/studio/index.html#downloads, and grab the link listed in the table for **Get just the command line tools** > Linux. Don't download anything, just copy the link.

```
sudo apt-get install unzip
```

here you paste the link you grabbed in the developer.android.com site

```
sudo -iu jenkins wget https://dl.google.com/android/repository/sdk-tools-linux-3859397.zip

sudo -iu jenkins mkdir android-sdk

sudo -iu jenkins unzip sdk-tools-linux-3859397.zip -d android-sdk

# this step is important to accept the Android SDK license

yes |sudo -iu jenkins android-sdk/tools/bin/sdkmanager --licenses
```

Step 3: Configure Jenkins

Then, log in to Jenkins and configure the Android SDK that you just installed: Open https://your-ci-server-name.com on your browser. You'll see something like this:

BUGFENDER
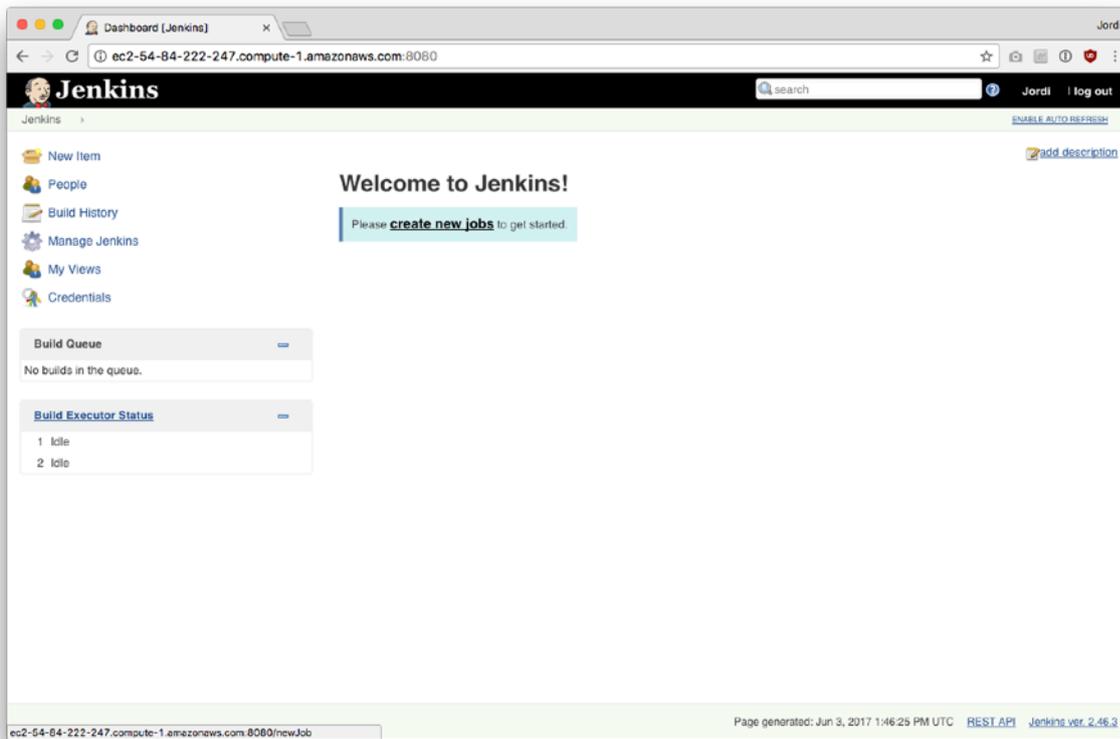
- Go to Manage Jenkins > Configure System

- Check "Environment variables"

- Add Name: ANDROID_HOME

- Add Value: /var/lib/jenkins/android-sdk

- Click "Apply" then "Save"

Step 4: Create an Android Job

Now go back to the home page. Click on **New Item**. Enter your project name and select "Freestyle project."

🐛 **BUGFENDER**
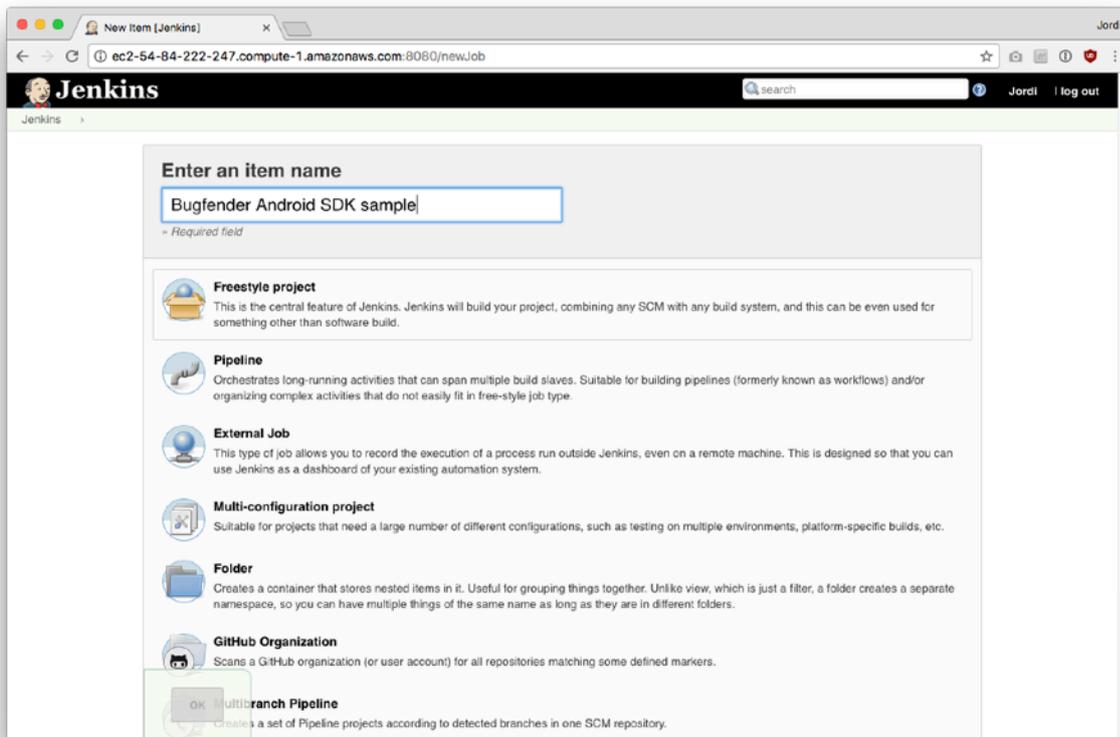
Step 5: Downloading Your Code to Jenkins

Next, you need to add a link to your repository so that Jenkins can download your code. Specify the Git URL of your repository in the Source Code Management section. As mentioned, we're going to use this sample repository: https://github.com/bugfender/BugfenderSDK-android-sample.git. If you have a Mercurial or Subversion repository, they work the exact same way. If you have a private repository, you can also create SSH keys to access your repository with the git protocol.

BUGFENDER
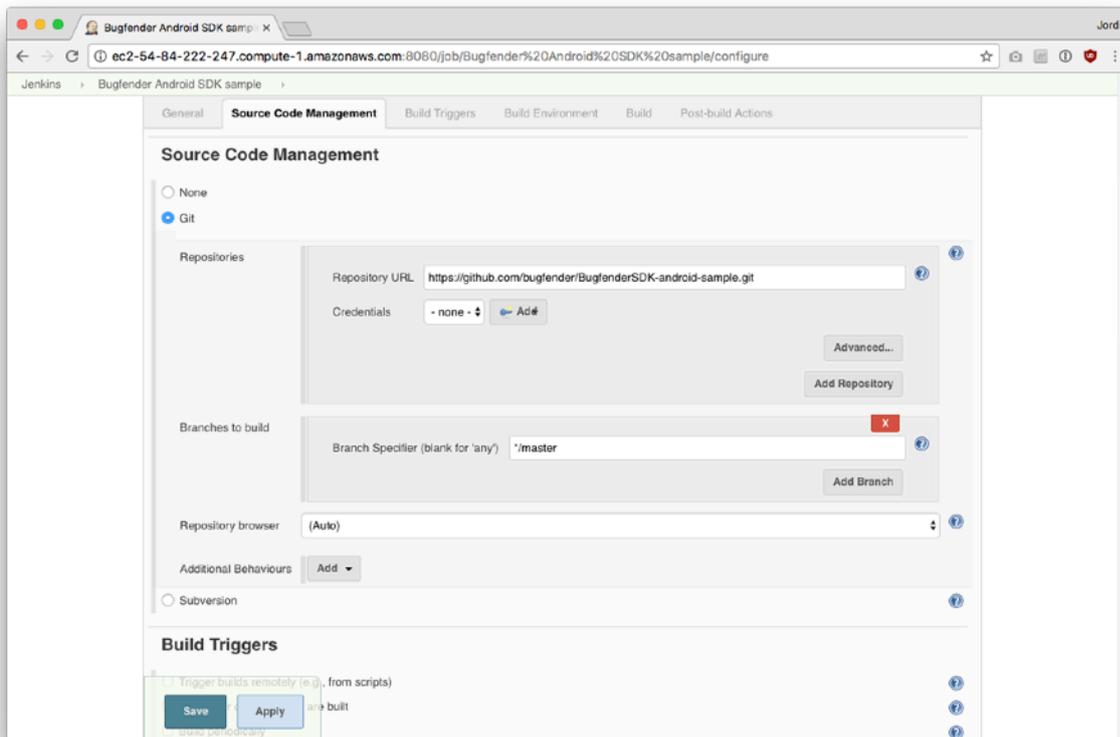
Step 6: Configuring Jenkins Build Triggers

You can also specify Build Triggers that will build the project automatically for you. It is best to use a hook in order to trigger builds automatically when someone pushes code to the repository.

For BitBucket, Gitlab and most Git providers, you can use "Trigger builds remotely" option. This will give a URL that you can configure as a webhook in your provider to automatically start a build.

For GitHub users, the easiest is to find and install the "GitHub plugin" (in Manage Jenkins > Manage Plugins) and a GitHub specific option will appear. This will install the webhook for you.

Step 7: Build

Once you have the source code in Jenkins, it's time to build. If your project has a Gradle wrapper, that's what you should use. In our case, we'll create an Invoke Gradle script build step like this:

If you have an Ant-based project, there is also an **Invoke Ant** build step or for maximum flexibility you can use the **Execute shell** build step.
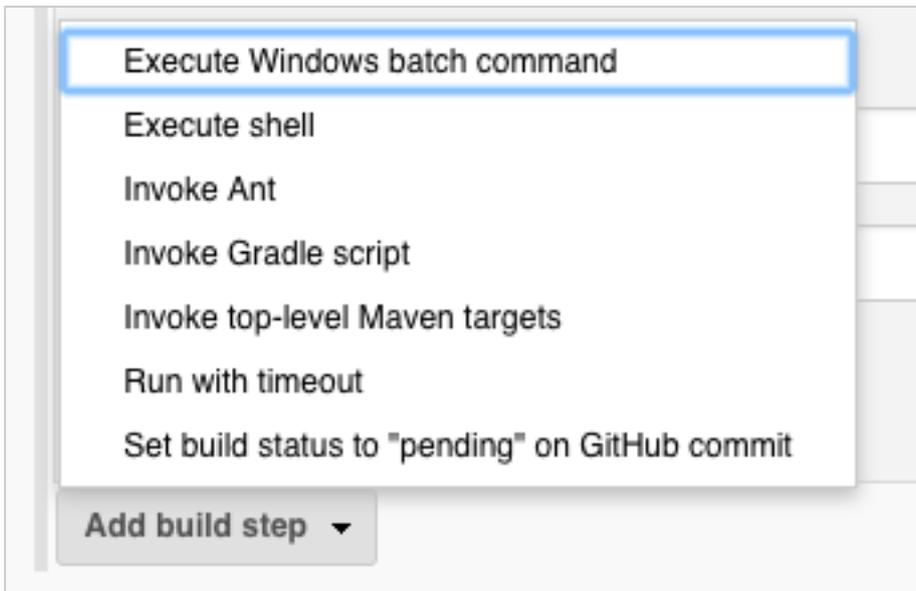
Test your build by pressing "Save."

And that's it. Your first Android project has been added to Jenkins. Now you may want to consider accelerating Gradle and setting up automated testing in Jenkins.

Accelerate Gradle (optional)

If you're running builds based on Gradle, you can save some build time by running Gradle as daemon. In order to get it running do the following:

```
mkdir -p /var/lib/jenkins/.gradle

echo org.gradle.daemon=true | sudo -iu jenkins tee -a /var/lib/
jenkins/.gradle/gradle.properties
```

BUGFENDER

Running Unit Tests

If you want to run unit tests, you can do so by adding another build step with the test Gradle task, like this:

BUGFENDER

Running Integration Tests

In order to set up UI tests, you will need to run an emulator. Even though Jenkins has an **Android Emulator Plugin**, we have found that it has not been updated and is no longer working with the latest versions of the Android SDK. Here is our suggested workaround:

Step 1: Download the emulator. In order to list the SDKs available, run:

```
sudo -iu jenkins android-sdk/tools/bin/sdkmanager --list --verbose
```

Quick note on selecting the right image for you: x86 based images run faster but also need hardware acceleration. You might have to enable KVM acceleration (`sudo modprobe kvm`) in your server and your server can not be a virtual machine. For this reason we're choosing an ARM image, which runs slower but works.

Step 2: In our case, we're looking to run our application on Android SDK level 25, so **system-images;android-25;google_apis;armeabi-v7a** seems suitable, then we install it this way:

```
sudo -iu jenkins android-sdk/tools/bin/sdkmanager 'system-
images;android-25;google_apis;armeabi-v7a'
```

Step 3: Once installed, we create an Android Virtual Device (an emulator instance) with it:

```
echo no | $ANDROID_SDK_ROOT/tools/bin/avdmanager -v create avd --
force --package 'system-images;android-25;google_apis;armeabi-v7a'
--name Android25 --tag google_apis --abi armeabi-v7a
```

Step 4: Then we use Supervisord to run the Android emulator as a system service, always in the background. In order to install it:

```
sudo apt-get install supervisor
```

Then, create a configuration file in **/etc/supervisor/conf.d/emulator.conf**:
[program:emulator]

```
command=/var/lib/jenkins/android-sdk/emulator/emulator -avd
Android25 -no-window -noaudio -no-boot-anim -accel on -ports
5556,5557

autostart=true

user=jenkins

environment=ANDROID_SDK_ROOT=/var/lib/jenkins/android-sdk
```

Step 5: Once this is done, restart supervisord to apply the changes:

```
sudo service supervisor restart
```

The emulator should start in the background. It might take 1-2 minutes. You will see the emulator device appear in the devices list when it's ready:

```
sudo -iu jenkins android-sdk/platform-tools/adb devices
```

Step 6: In the Jenkins job, add an Execute shell build step like this:
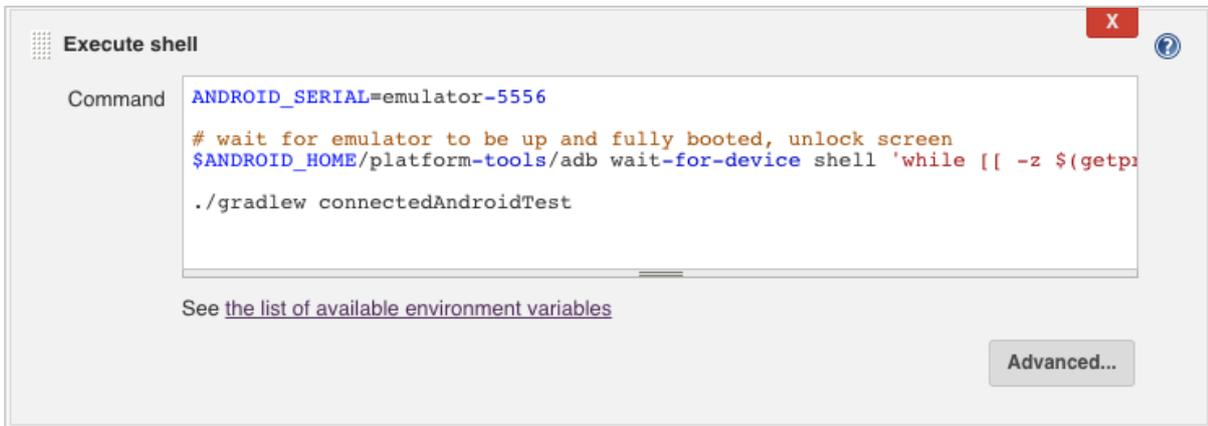
Code:

```
ANDROID_SERIAL=emulator-5556

# wait for emulator to be up and fully booted, unlock screen

$ANDROID_HOME/platform-tools/adb wait-for-device shell 'while [[ -
z $(getprop sys.boot_completed) ]]; do sleep 1; done; input
keyevent 82'

./gradlew connectedAndroidTest
```

Suggestions for Next Steps

There are more options like running an Android emulator, installing the application, and running various tests on it.

In addition, there are some really interesting plugins that you can play with:

➔ Android Emulator Plugin — At the moment, this plugin is not properly working with latest Android SDK. But our best guess is that they will fix it and it will be working again soon. As an alternative, you can use an older SDK version.

➔ Android Signing Plugin

➔ Google Play Android Publisher Plugin

BUGFENDER

How to Add Your First iOS Job to Jenkins

In order to build iOS applications you will need to run the server on a macOS machine. You could install Jenkins directly on a Mac computer, but our preferred way is to have a small Mac mini machine* in our office devoted to these tasks.

Even though there are some server hosting options for macOS, they are not as commonly used as Android hosting options. Our solution is to run Jenkins in a Linux machine and have a macOS worker machine for the iOS builds.

(*We mention a Mac mini because it's the cheapest macOS machine you can get. In 2017, new models started at around $499. But if you have any other Macs laying around, such as an old MacBook with 1GB of RAM or more, that can work as well.)

BUGFENDER

How to Add a Mac Node for iOS Builds

Follow these steps to add a Mac node as a worker node to a running Jenkins server. If you've already done this step, skip to "Configuring iOS jobs" below to start adding a iOS build to Jenkins.

Please note before we begin: you do not need a public IP address or open ports on the Mac node side, so it's a perfectly suitable setup to have the Mac machine in your home or office, sitting behind a NATed network.

Step 1: Open https://your-ci-server-name.com on Your Browser

You will need to open a port for JNLP. JNLP is a Java protocol to serve applications to be executed elsewhere. Your worker node will download the Jenkins worker application using this protocol.

Step 2: From the Dashboard Go to Configure Jenkins > Configure Global Security

Specify a **Fixed** port. In the example below we chose 8081, but you can choose any port between 1024 and 65535. Make sure this port is open in your firewall.
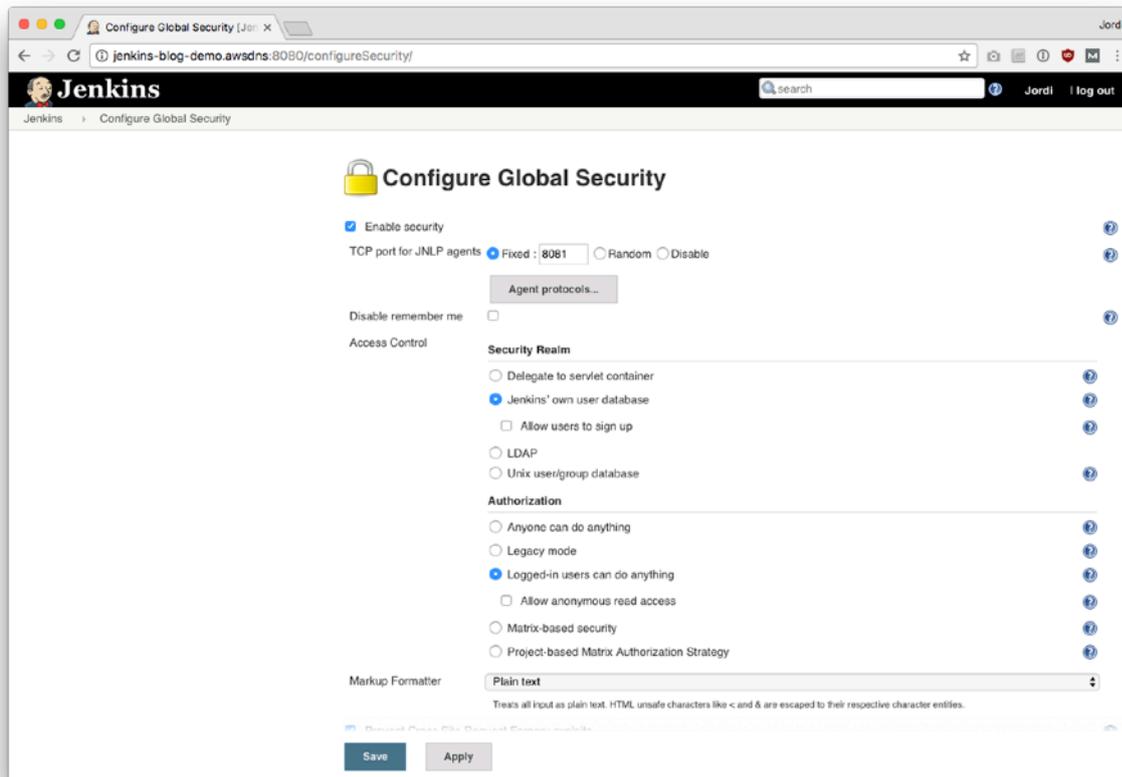
Step 3: Create a New Node In Jenkins > Manage Nodes
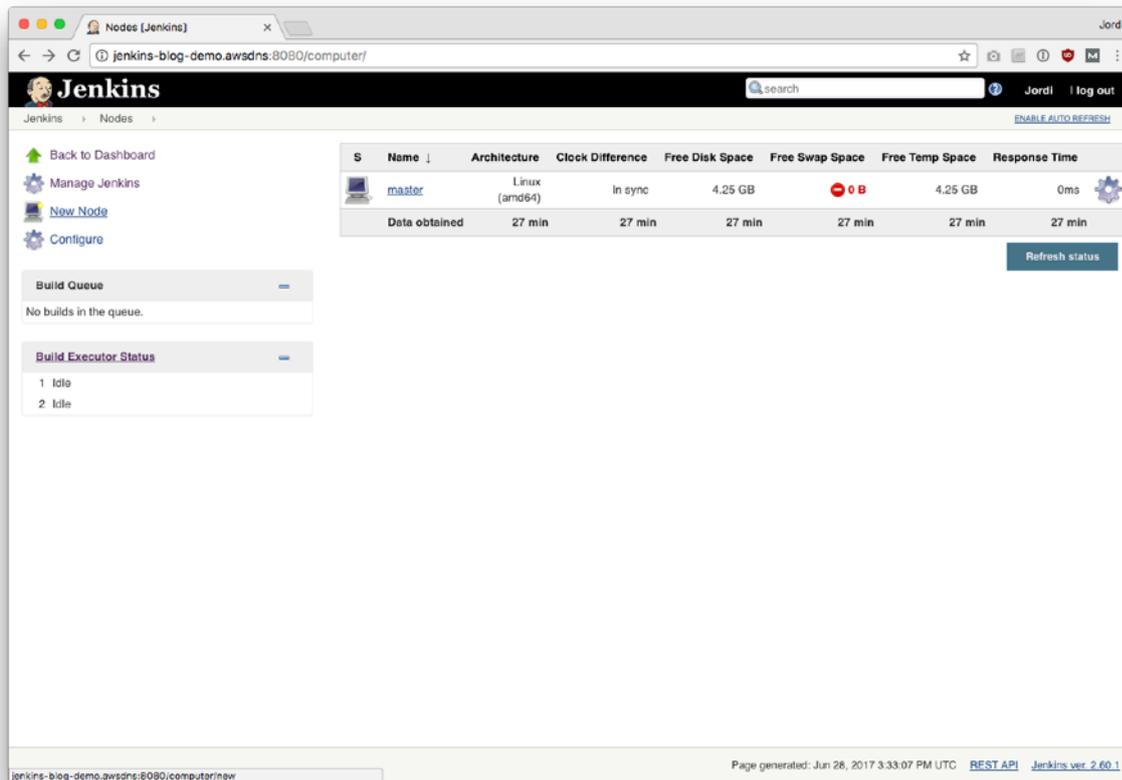
Select **New Node**.

- Give it a name, for example macmini. It can be anything.

- Enter a Remote root directory: /tmp

- Choose Launch method: **Java web start**

- Add a Label: **mac**. (This label will be important when you need to specify where jobs should be executed each time you configure them. Using this label, you can make sure Mac projects exclusively run on your mac machine. More on this below.)

BUGFENDER

After saving you'll see the newly added node in the node list.

Step 4: Open the Node You Just Created and You Will Find a Command

Copy and paste this command in a terminal in your slave machine.
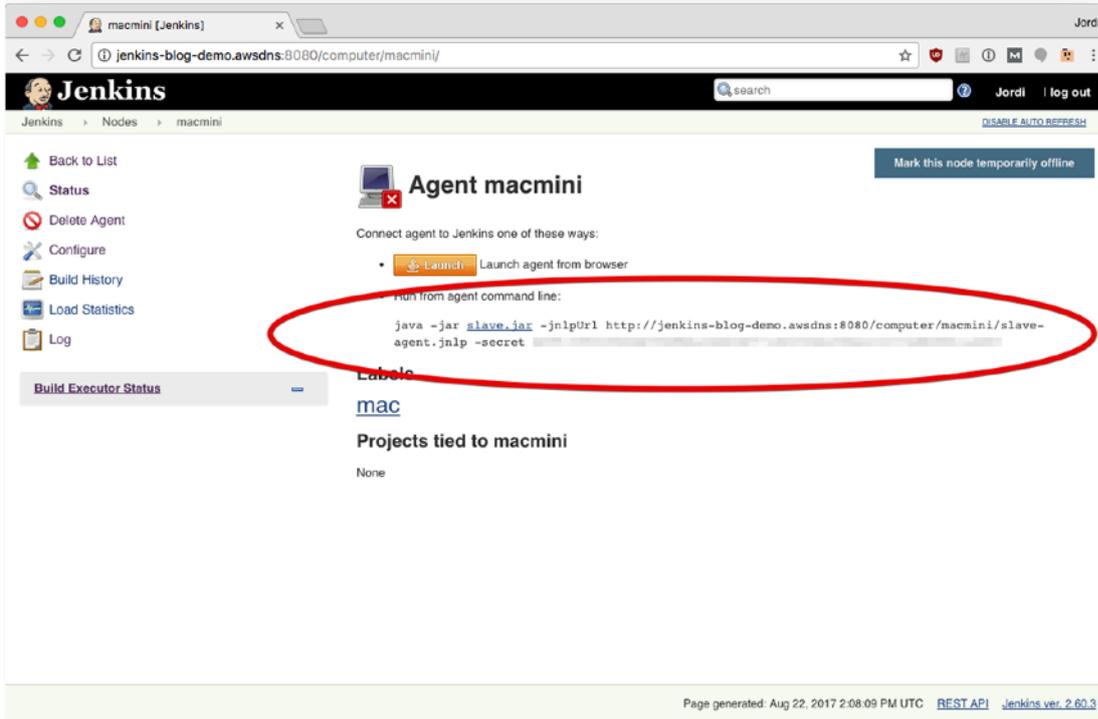
After a few seconds you'll get the node up and running in your node list.
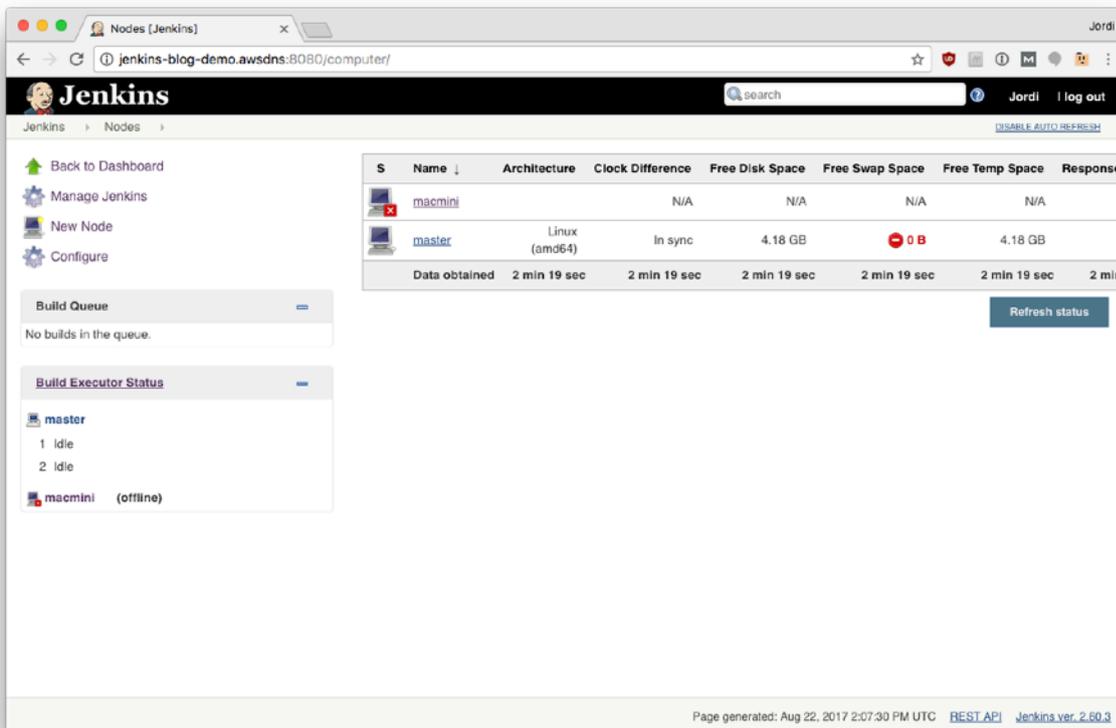
Now your node is ready to accept build jobs!

Step 5: Download Xcode

Since you are planning to build iOS projects, you will need to download **Xcode**. You can find it in the app store.

Command line tools: once you have Xcode installed, open a Terminal and type:

```
xcode-select --install
```

Configuring iOS Jobs

Now that you have a Mac worker machine, you're ready to start building iOS apps.

Step 1: Open https://your-ci-server-name.com on your browser and select **New Item**

Specify the name of the new job and select **Freestyle project**:
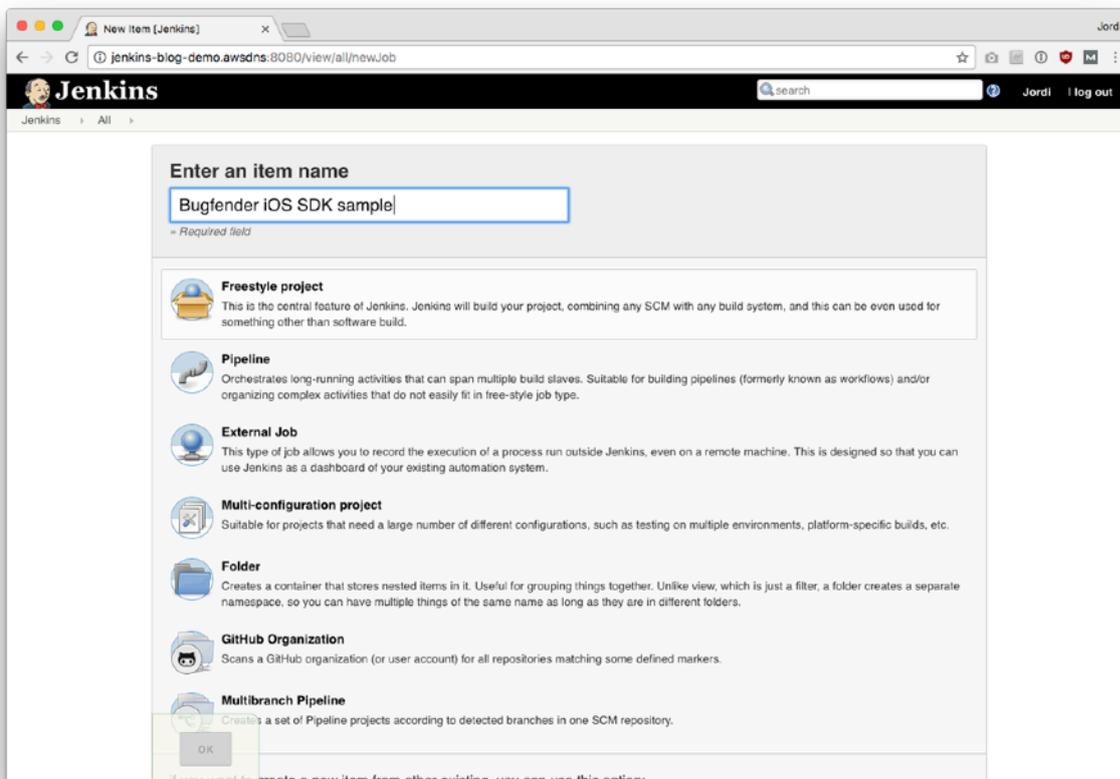
Step 2: Make sure your build is running exclusively on the Mac machine.

In order to do that, in the **General** tab, select "**Restrict where this project can be run**" and write mac as label expression (this label matches the one we chose for the Mac worker in the node configuration above).

Step 3: Under Source Code Management, specify the URL of your repository. For our example, we're building the Bugfender SDK sample app:
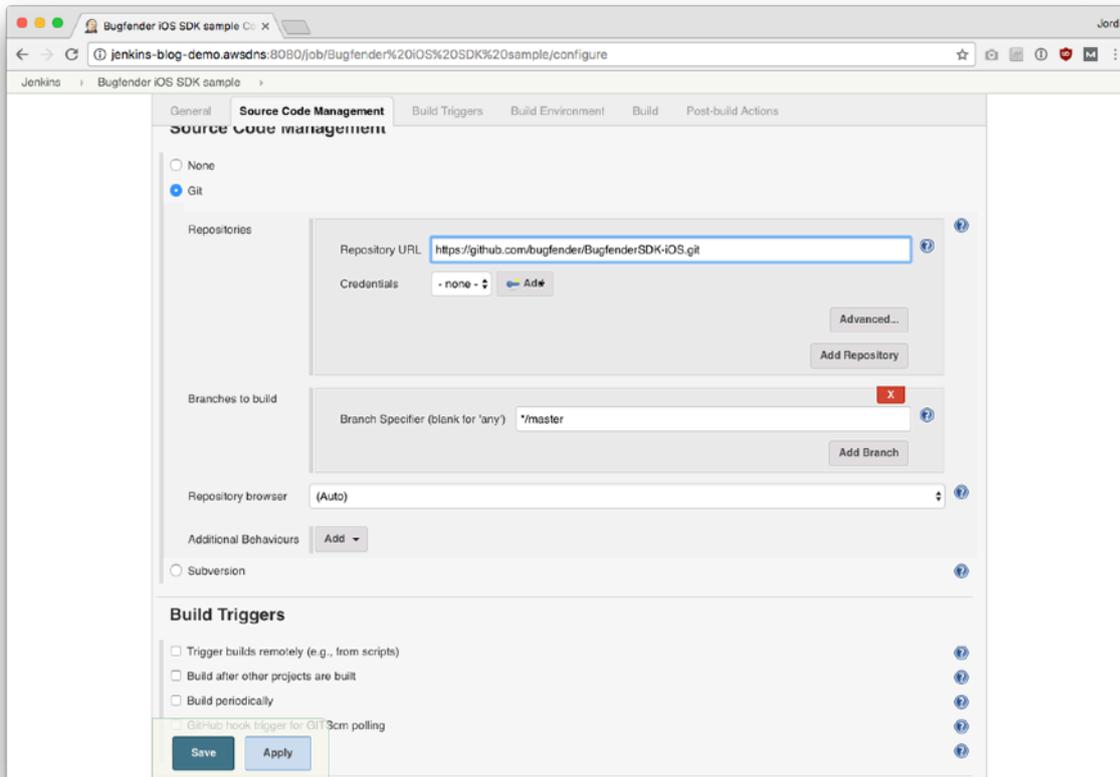

Step 4: Configuring automated build triggers.

You can select automated triggers for your builds if you don't want to have to manually trigger them. Typically, you should configure it so that a new build is done automatically for every commit or a build is done periodically (eg. nightly builds):
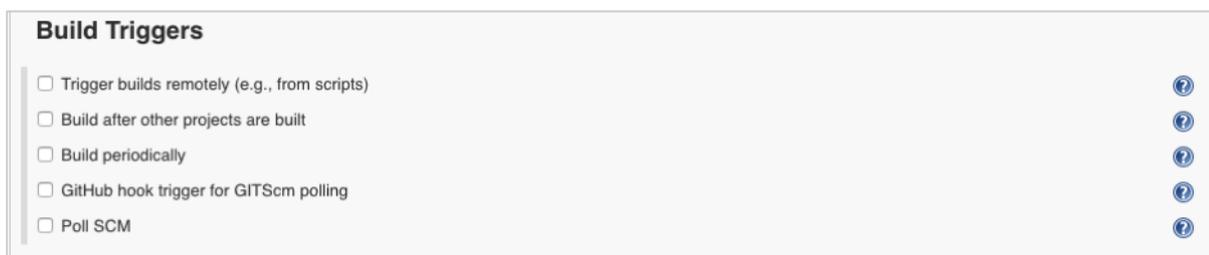
Step 5: Once the code is downloaded, we need to specify how to build it.

BUGFENDER

In the **Build** section, we will **Add a build step** of type **Execute shell** and we will execute xcodebuild like this:
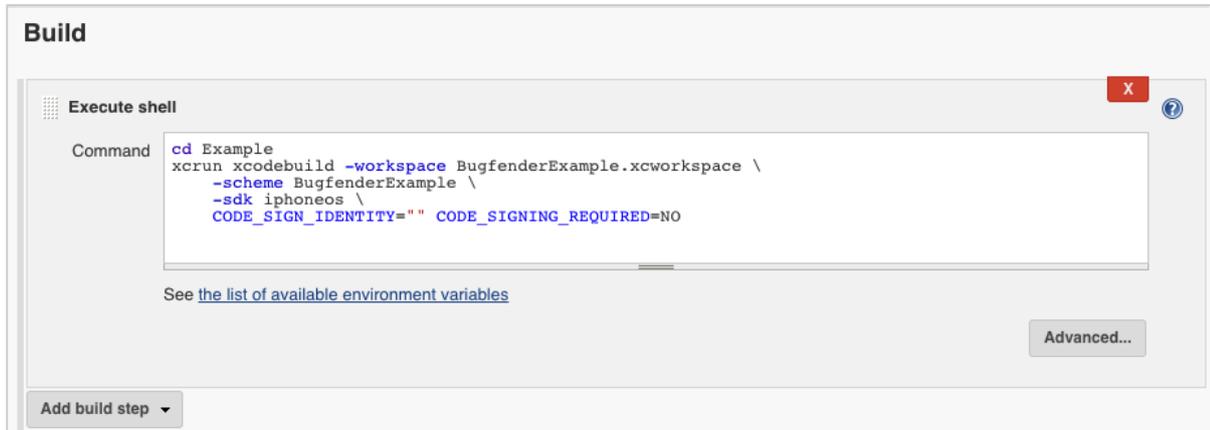
cd Example # in our example, the project is not in the root of the repository

```
xcrun xcodebuild -workspace BugfenderExample.xcworkspace \

    -scheme BugfenderExample \
    -sdk iphoneos \
    CODE_SIGN_IDENTITY="" CODE_SIGNING_REQUIRED=NO
```

This will trigger a build of the application for iOS devices, similar to when you use the **Product > Build for Running** in Xcode.

Make sure you add the **CODE_SIGN_IDENTITY=""** **CODE_SIGNING_REQUIRED=NO** bits in order to prevent Xcode from trying to sign the built application, otherwise you will need to configure your provisioning profiles and developer certificate in that machine, which makes everything a bit more complicated.

BUGFENDER

Adding Tests

If you want to run tests, you can set them up with another build step:

Code:

```
cd Example # again, we need to move directory

mkdir -p output # this directory will contain the output of the
tests

xcrun xcodebuild -workspace BugfenderExample.xcworkspace \

    -scheme "BugfenderExample" \

    -sdk iphonesimulator \

    -destination 'platform=iOS Simulator,name=iPhone 6,OS=10.0' \

    -derivedDataPath './output' \

    Test
```

And that's it! You're ready to build and test our iOS sample application:

BUGFENDER
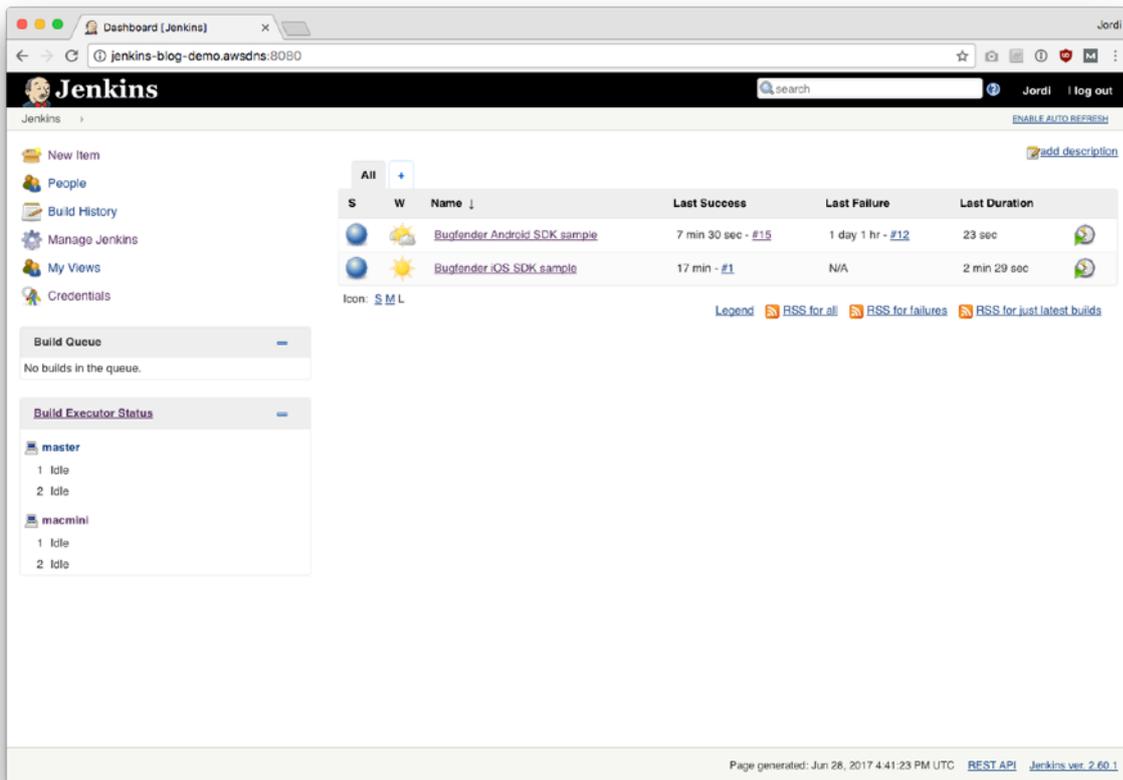
Looking for more information about developing apps? Check out:

● Top Devices for Testing iOS Apps

BUGFENDER

Closing Words

Keys to Sustainably Managing Jenkins

Though Jenkins is practical, it hasn't been easy.

In the beginning, some of our team members tended to ignore Jenkins because they weren't used to it. As developers, we recognize that this is understandable. We have gradually adopted it for more Android and web projects, and everyone has increasingly gotten more accustomed to it. Now it's a tool we rely on for our day-to-day work.

To increase usability, we have made <u>three</u> main improvements:

1. We originally ran Jenkins on an old MacBook Pro that was gathering dust in the office. It turns out a laptop is not a great machine for running a continuous service like this one because it heats up pretty quickly. So now we run it in a **cloud server for the web and Android workloads** with a **Mac Mini as worker machine for the iOS and macOS projects.**

2. We use git-flow as a way to structure and collaborate in the git repository. Together with Jenkins, this helps us find out earlier if something is not working properly. We can view each functionality under development separately so we can solve issues before we put everything together for a release.

3. We also use Jenkins for continuous delivery. Once we merge something to the "develop" branch (again using git-flow conventions), the beta version of the app is built automatically. Once we merge to the master, a release version of the app is built automatically, ensuring every build

BUGFENDER

has passed all CI tests and is built in a consistent manner, leaving less room for human error.

In the future we would also like to integrate the results of Jenkins builds back into Bitbucket using the "build status" API.

Tips for Getting Started Today

First, we recommend you start small and grow your setup over time. Start with only 1-2 things that bring a lot of value to your business.

For example, a good way to start is by making the app builds automatically and publishing them for testing (somewhere like Google Play Beta, TestFlight or TestFairy).

This seems simple, but already provides a lot of value by standardizing the way you build your app, removing the human risk in an error-prone task, and making beta builds available to everyone in your team.

Second, make sure to allocate some time in your team schedule for running a CI system and writing tests that work. This might or might not be right for your team, so it will require some thought and planning. Don't just don't do it because it sounds cool or sounds like "the right thing to do".

If you understand the benefits, you will be willing to put up with the burden of fixing it when it doesn't work (and for sure this will eventually happen). If you don't, you will probably end up giving up and then the investment you made by setting it up in the first place will be lost. This can leave a bad impression.

Lastly, don't forget that setting up a CI/CD environment is all about the bottom line. In the long-term, Jenkins is an investment that will save your company money by significantly speeding up the build process. If you find yourself trying to convince your team or your manager—or the suits!—that Jenkins is the optimal solution for your devops team, don't forget to crunch the numbers and show how much money it will save the company in the long run.

That's it! If you have questions or want to reach out, we would love to hear from you. Chat with us directly on our website or email us at support@bugfender.com.

BUGFENDER