Good afternoon everyone –
Thanks for being here…..

We're going to be looking at Jenkins
-- brief intro
-- how to care for it

My name is Jeff McKenzie,
And I am a Practice Manager for App Dev and Infrastructure
At Insight Digital Innovation in Columbus Ohio

[How many Jenkins, types of jobs?]

If you're interested in automation,

You want to interact with Jenkins as little as possible.

So we are going to be looking at a case study

To demonstrate ways we can scale Jenkins in a better way.

And when I say case study….

=========
Logo from https://jenkins.io/artwork

I’m talking about Santa

--case study of all case studies

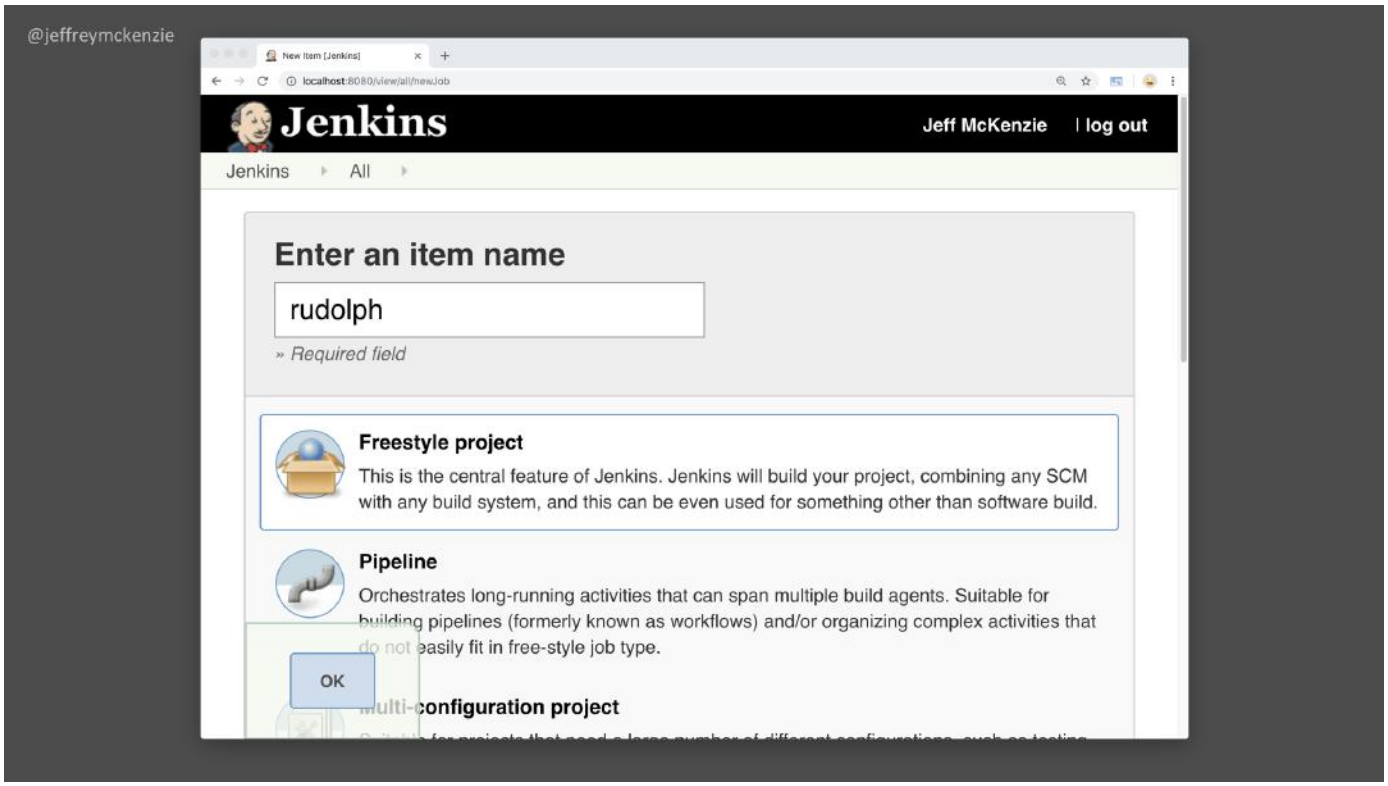
-- you talk about someone who needs to scale….

Specifically, communicate with his reindeer

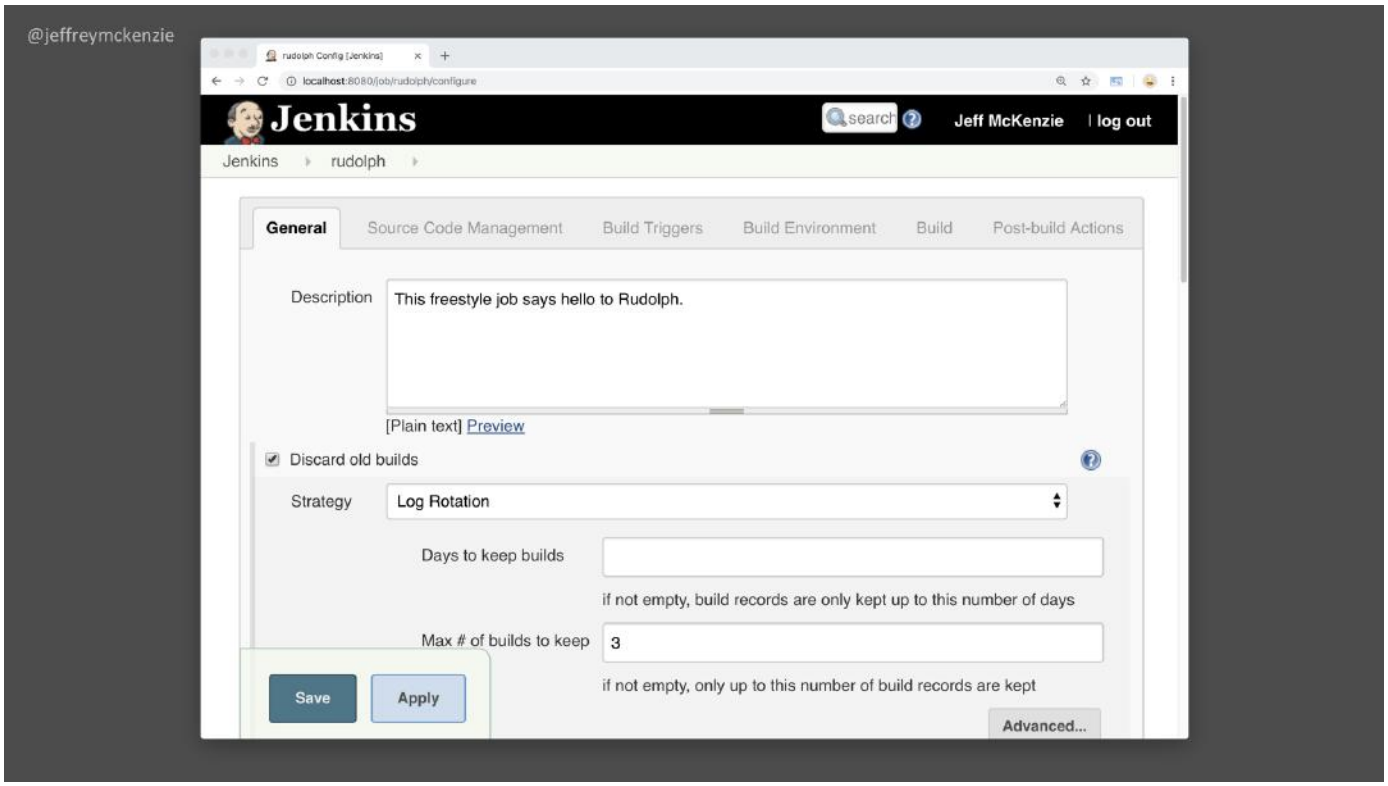
So let’s look at an example of what Santa’s going to do here.

-- create new freestyle job called Rudolph…

=========
Logo from https://jenkins.io/artwork
Created by https://twitter.com/ks_nenasheva

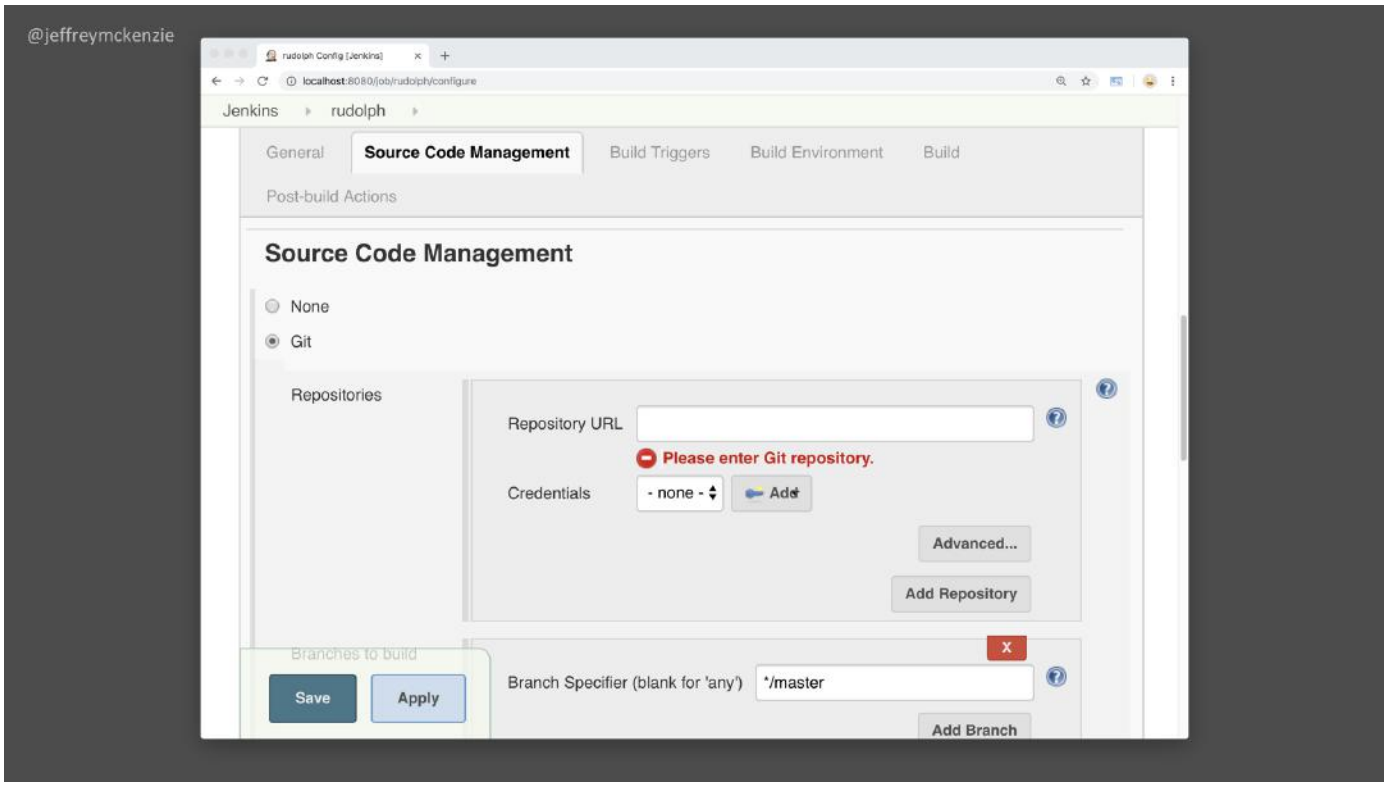

@jeffreymckenzie
New Item [Jenkins]
localhost:8080/view/all/newJob
Jenkins
Jeff McKenzie
| log out
Jenkins
All
Enter an item name
rudolph
» Required field
Freestyle project
This is the central feature of Jenkins. Jenkins will build your project, combining any SCM with any build system, and this can be even used for something other than software build.
Pipeline
Orchestrates long-running activities that can span multiple build agents. Suitable for building pipelines (formerly known as workflows) and/or organizing complex activities that do not easily fit in free-style job type.
OK
Multi-configuration project

Click OK…

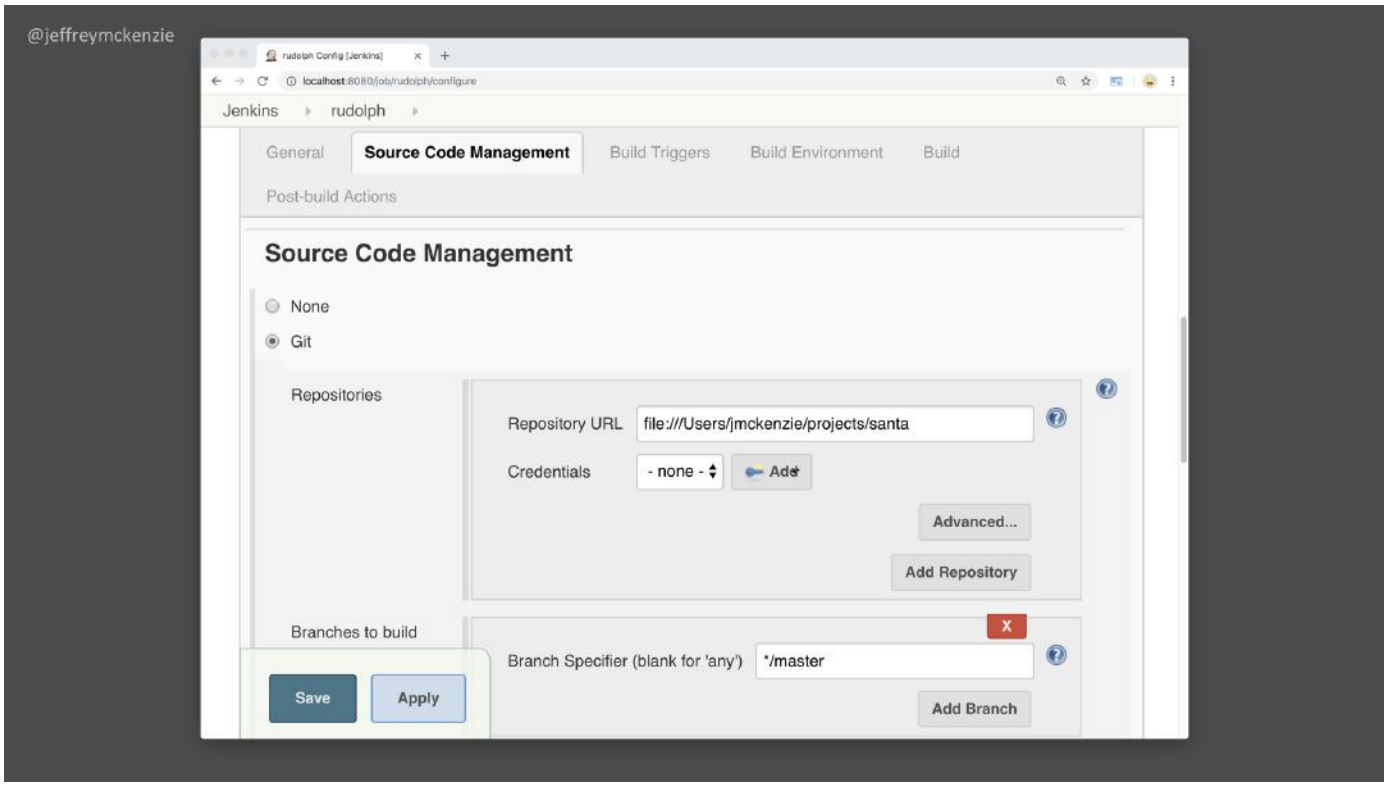


-- description, says hello to Rudolph

-- keep last 3 builds



Under the source code management tab, select Git

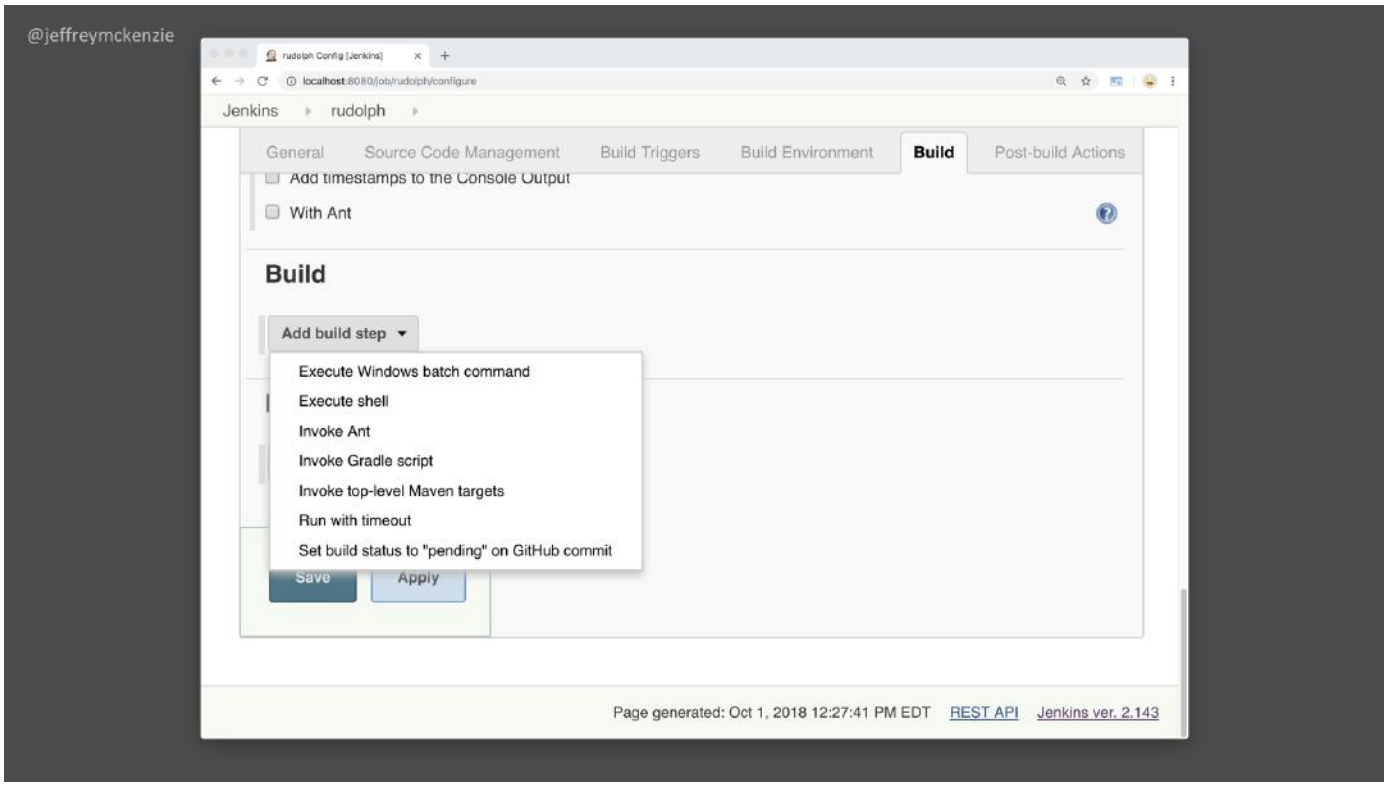


I'm just going to point this to a local git repo

That I have in my home directory

And it's pulling from the master branch

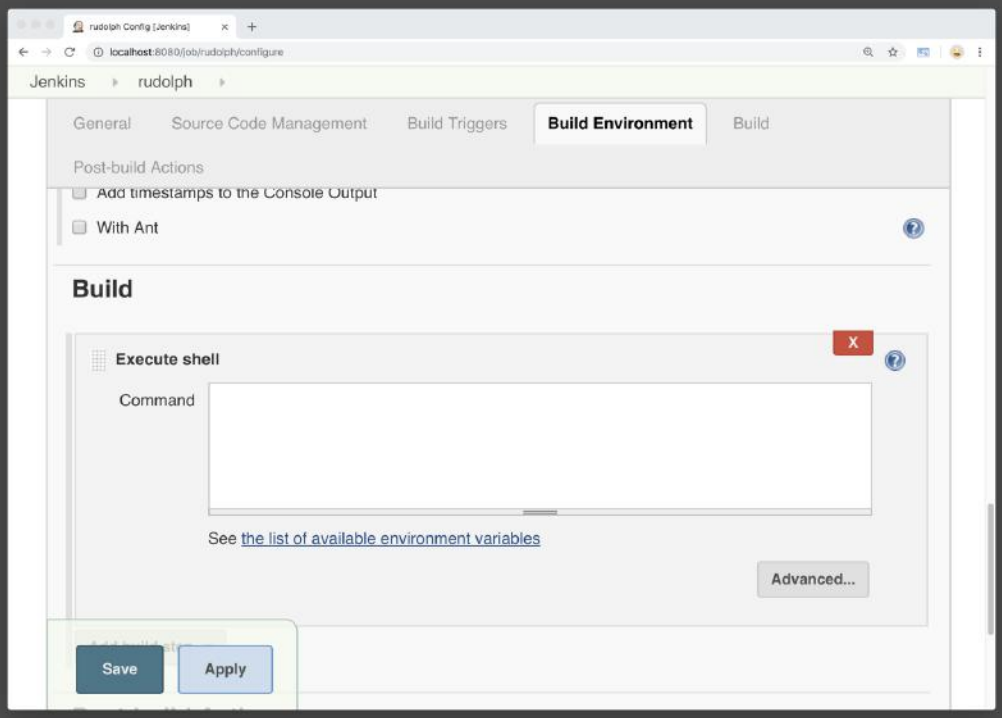
[Git users here?]

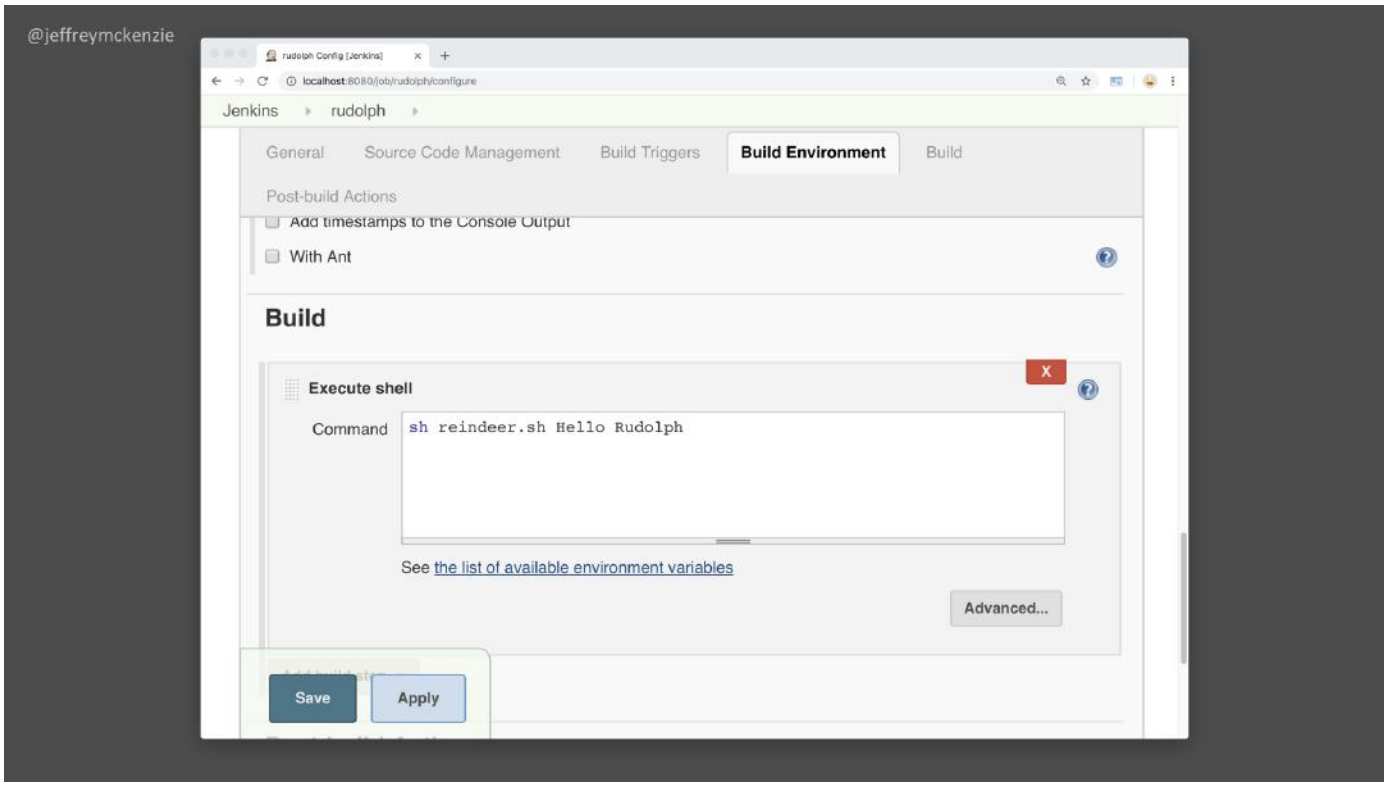


On the build tab, I'm going to choose

The Execute shell build step…

@jeffreymckenzie





And put the following command in there…

Sh Reindeer.sh Hello Rudolph.

This tells Jenkins to run a shell script called reindeer.sh

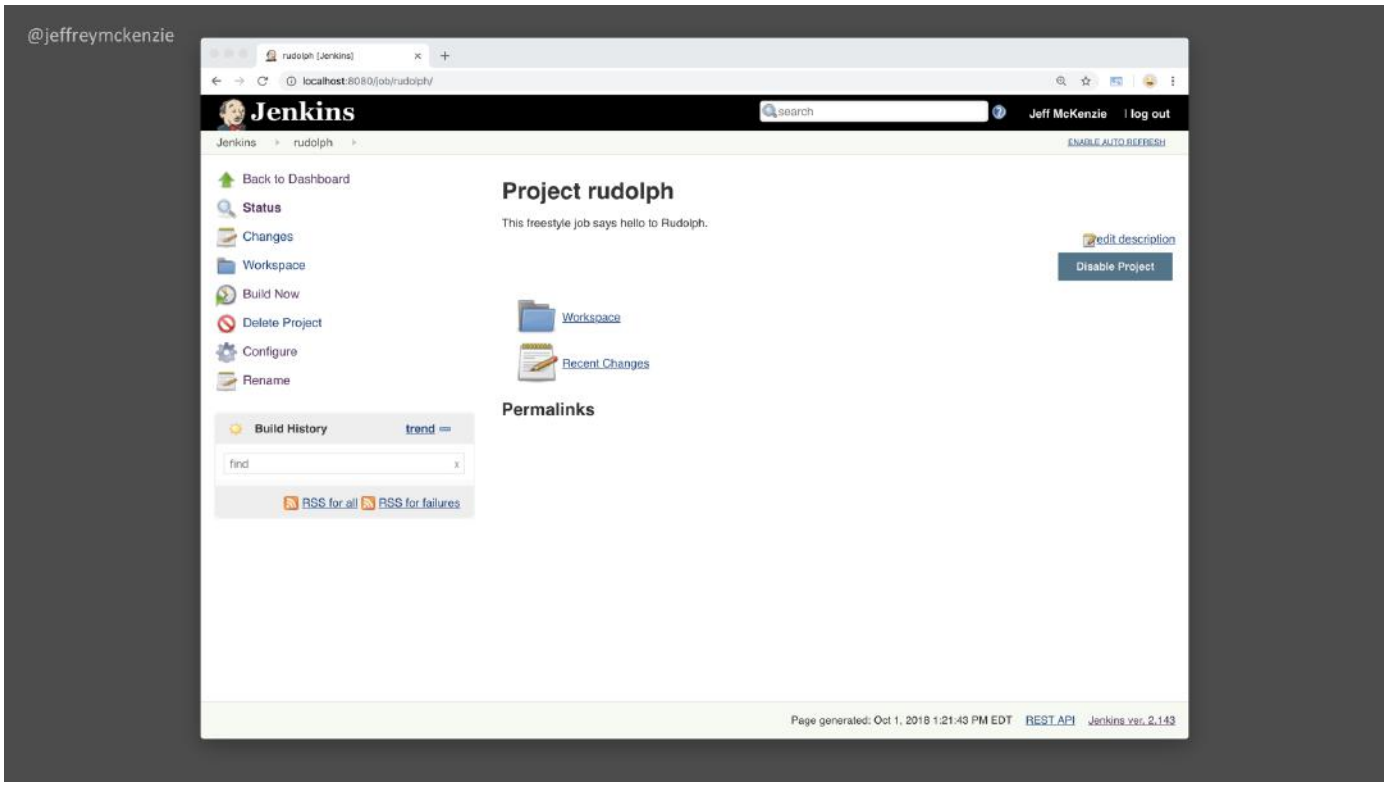
That’s in my git repo, so let’s take a look at that real quick.

@jeffreymckenzie

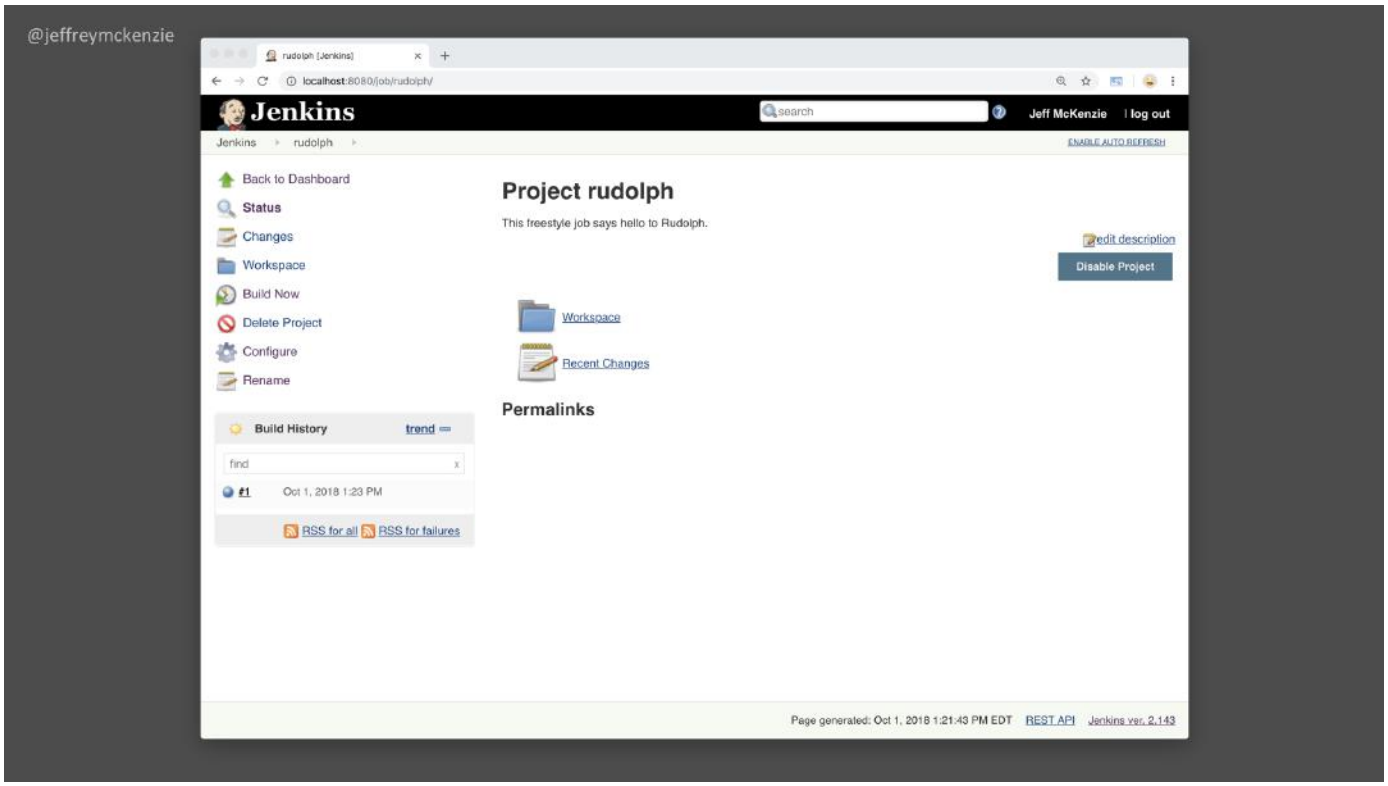
```
#!/bin/bash
greeting="$1"
reindeer_name="$2"
if [ -z "$greeting" ]
then
    greeting="Hi"
fi
if [ -z "$reindeer_name" ]
then
    reindeer_name="Reindeer"
fi
echo ================================
echo =
echo = ${greeting} ${reindeer_name}!
echo =
echo ================================
```

So this sets two variables –
Greeting, and reindeer name,
Which get passed in from Jenkins.
If nothing gets passed in for either variable,
Then the script sets defaults.
Then the result gets written as output.

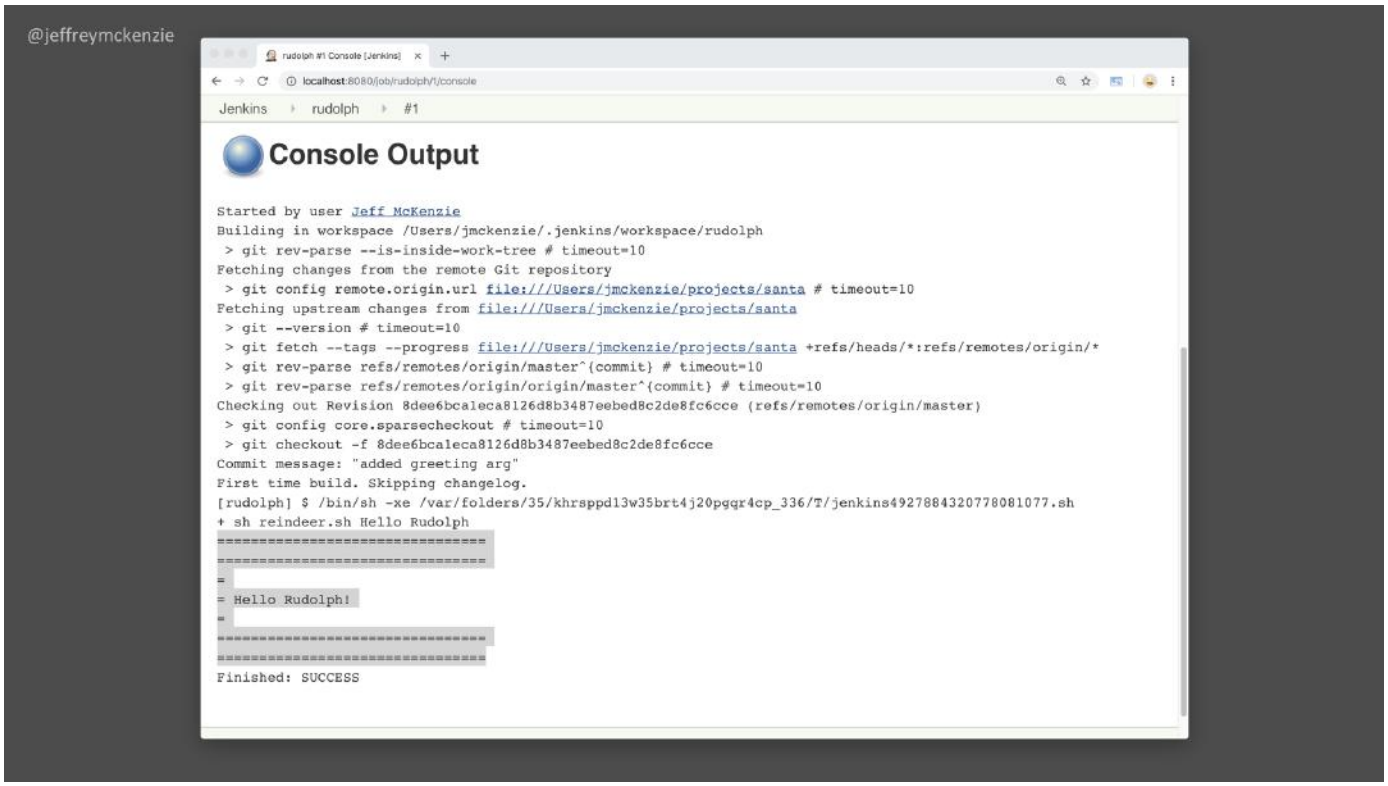
So we will save this, go back to the project screen



And click “build Now”



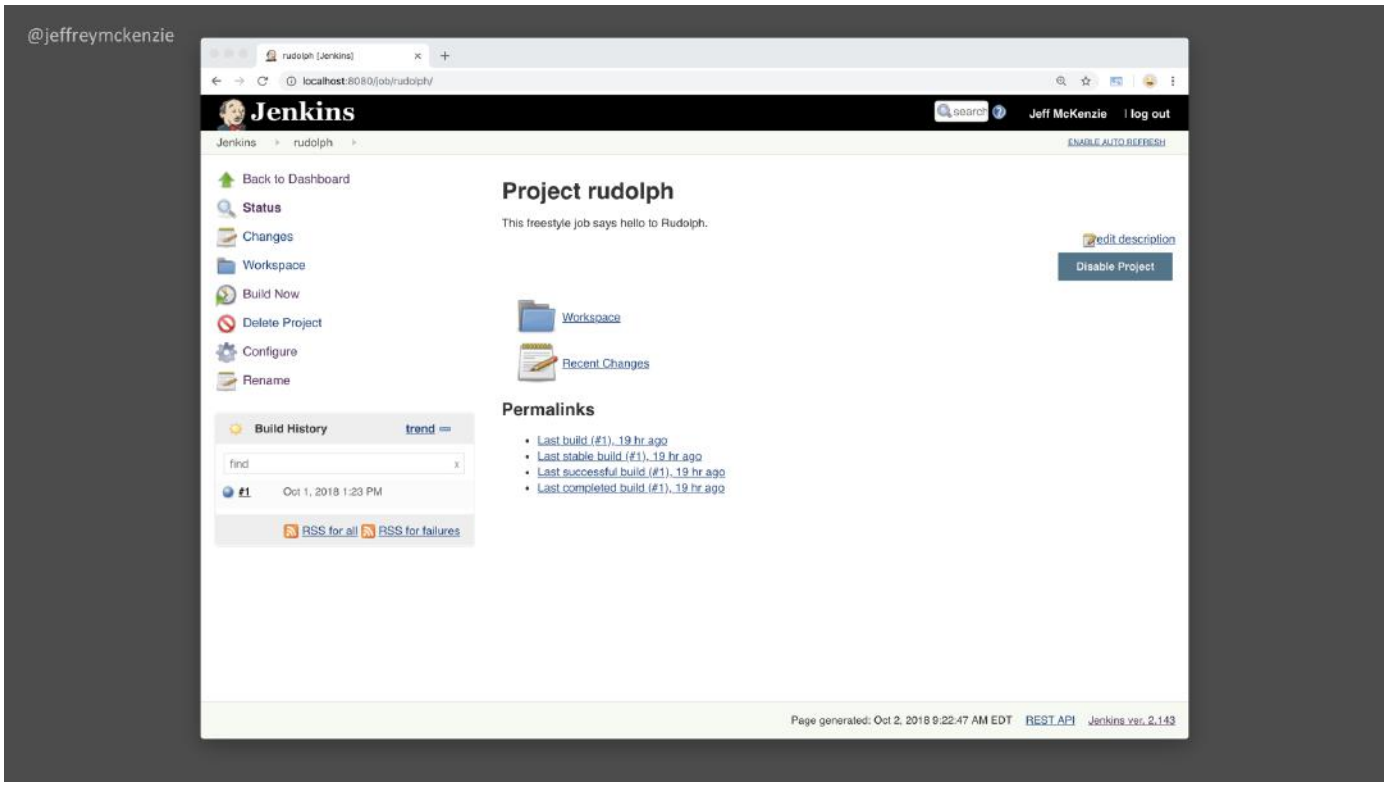
And we have a success – let's look at the output.



There we have the result of the script as it's run.
So there's one job for one reindeer.

-- contrived example, but basic
-- does have the basic elements:
    pulling code from source control
    and doing something with it

--not concerned as much with what it does as
    how to scale and manage these types of jobs



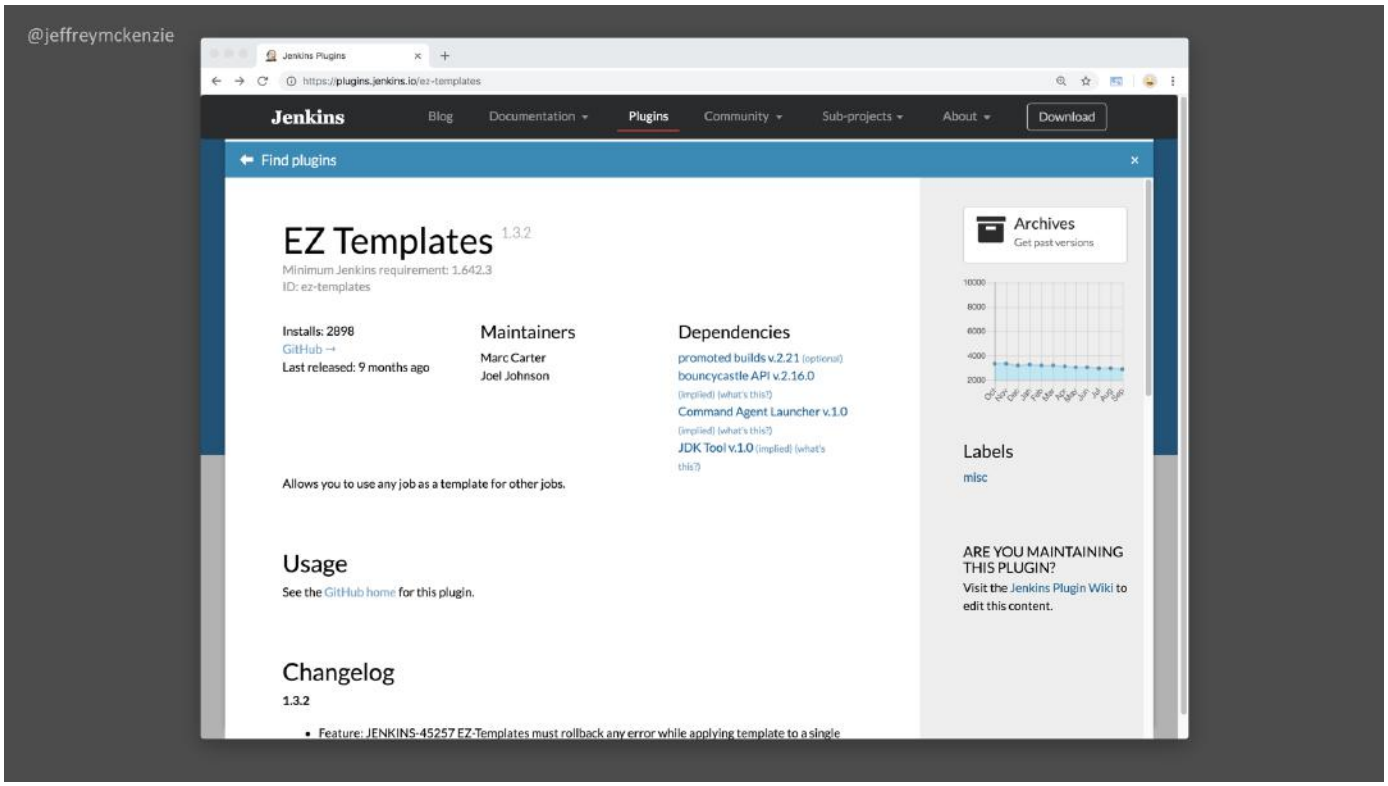
Here's our Rudolph project.
Let's say we copy this job 9 times for a total of 10 reindeer,
And we want this to say something other than Hello as a greeting?

[change each one individually….]

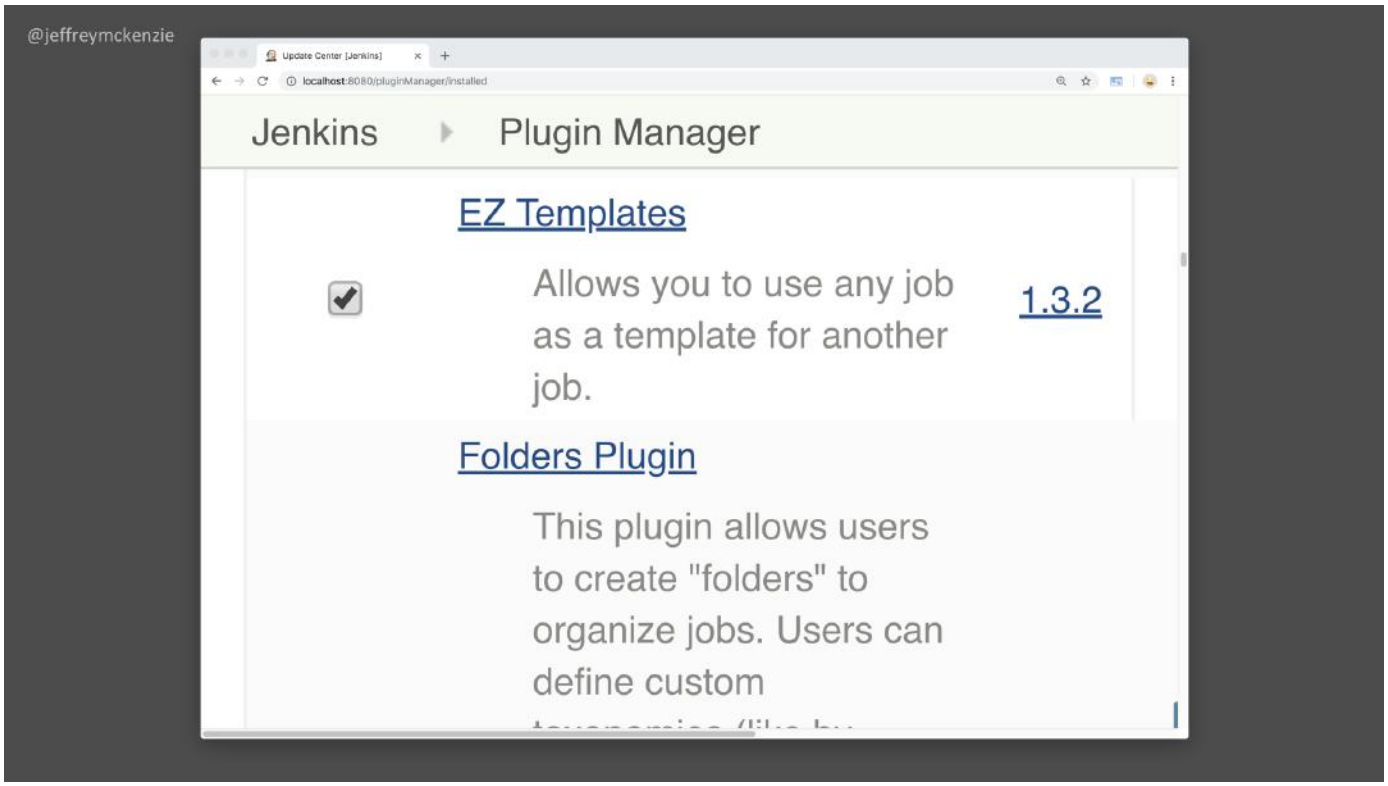
You get spaghetti automation that ties your hands together.

There’s a better way to do this….

==========
https://commons.wikimedia.org/wiki/File:Homemade_fresh_pasta_(spaghetti)_by_Camille_(Unsplash).jpg
By davide ragusa davideragusa (https://unsplash.com/photos/FwiLgvi-2Do) [CC0], via Wikimedia Commons



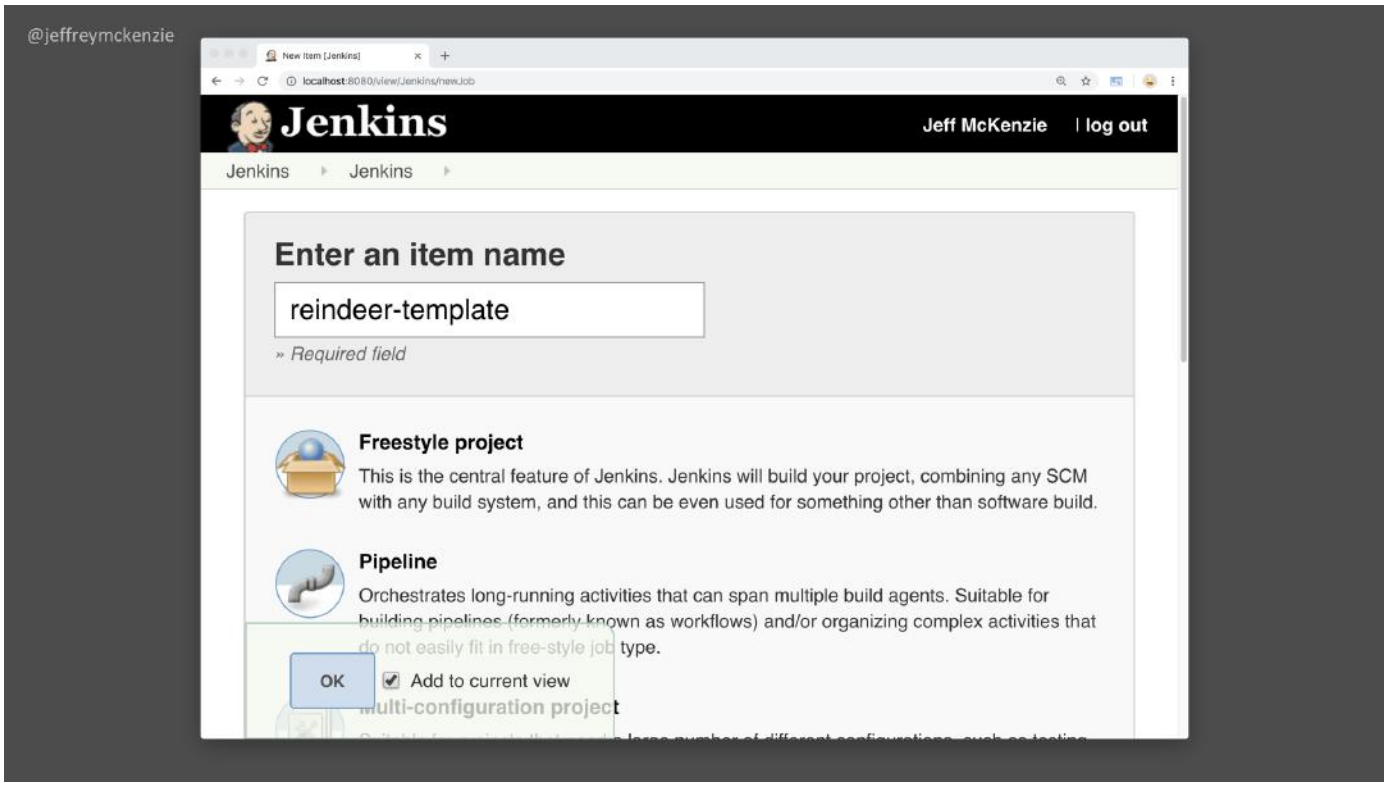
And one way is to use a plugin called EZ Templates
Here's the official plugin page for it,
There's also a git repo for it
the plugin itself is available in the Manage Plugins
section of Jenkins



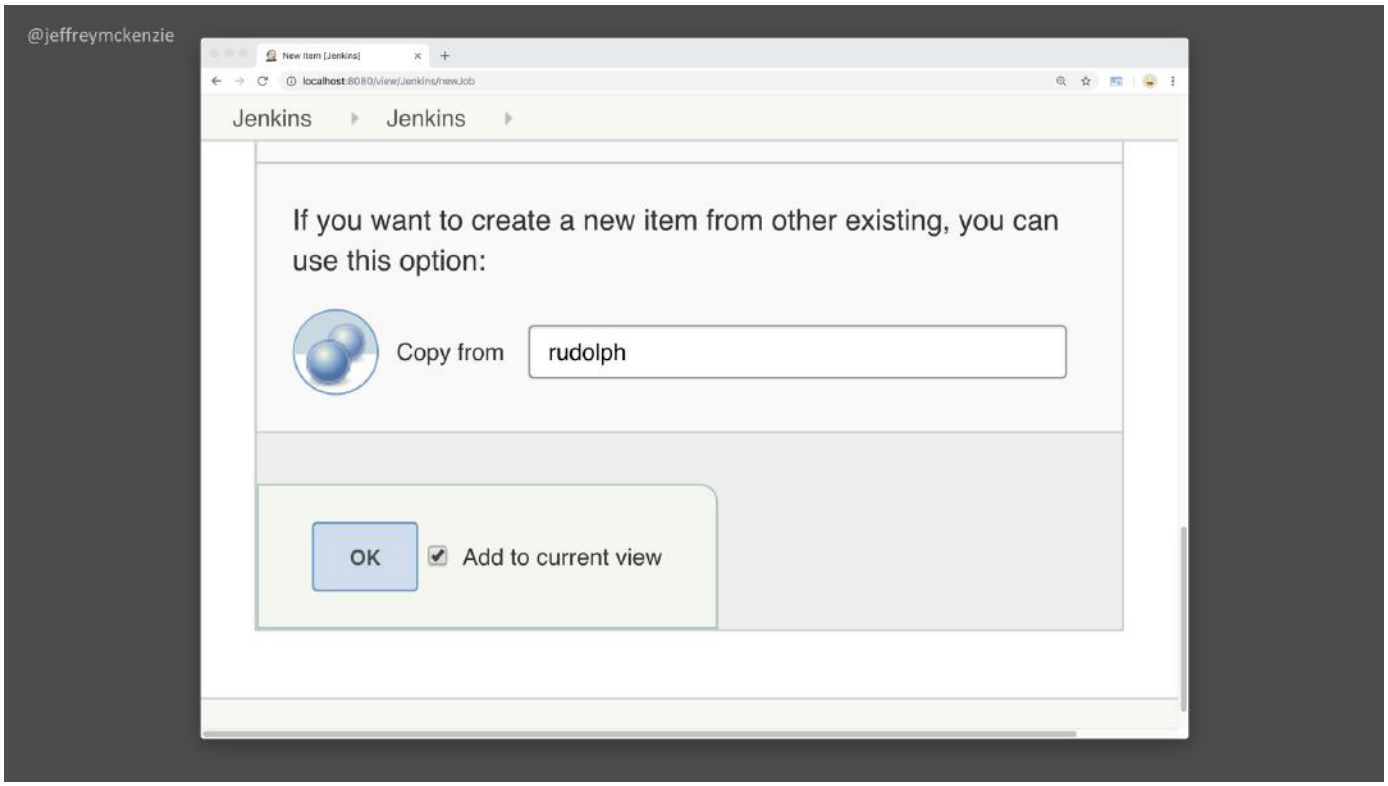
So you just need to select and install it

As it describes, this allows you to take any job
And use it as a template for other jobs.

Let’s take a look at how it might help us scale in this instance.

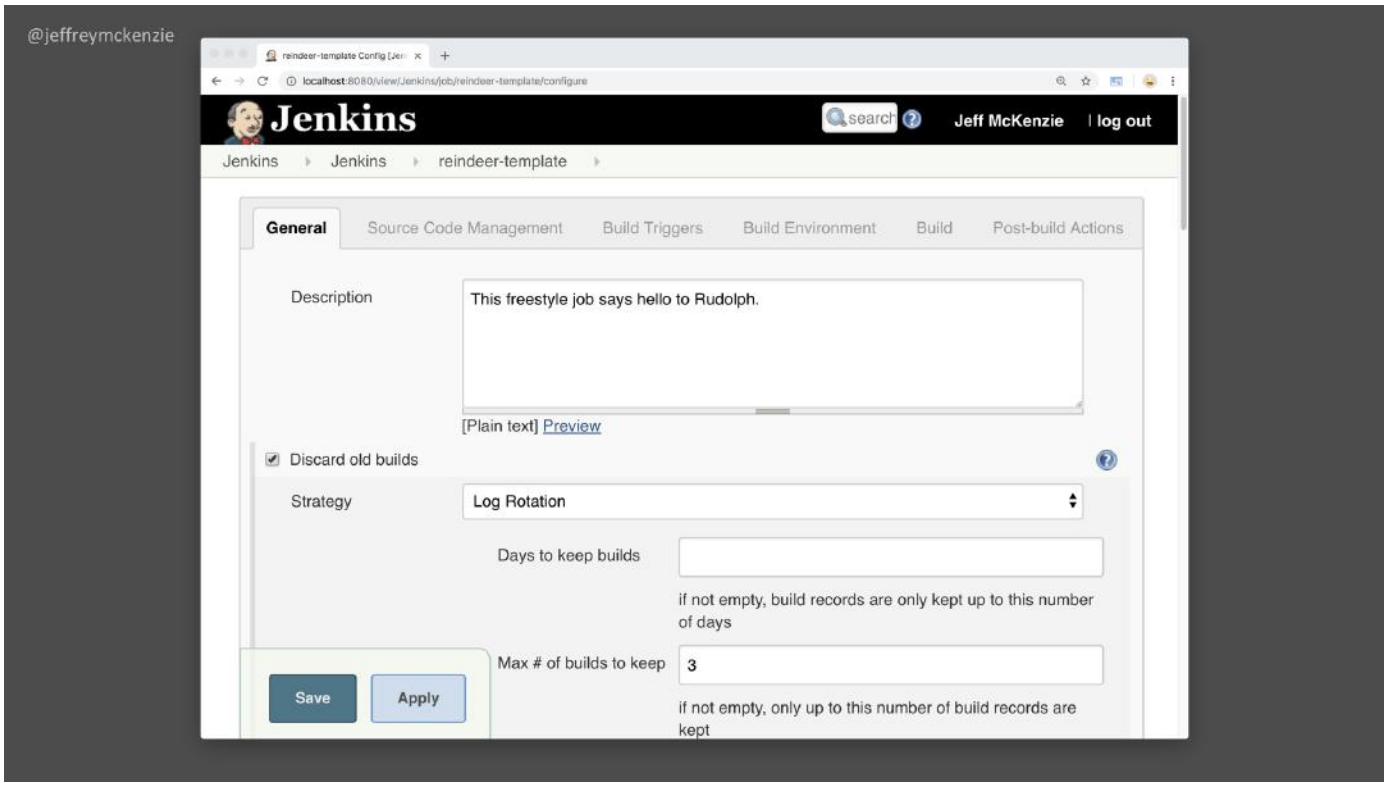


Create new freestyle job
Let’s call it Reindeer Template…

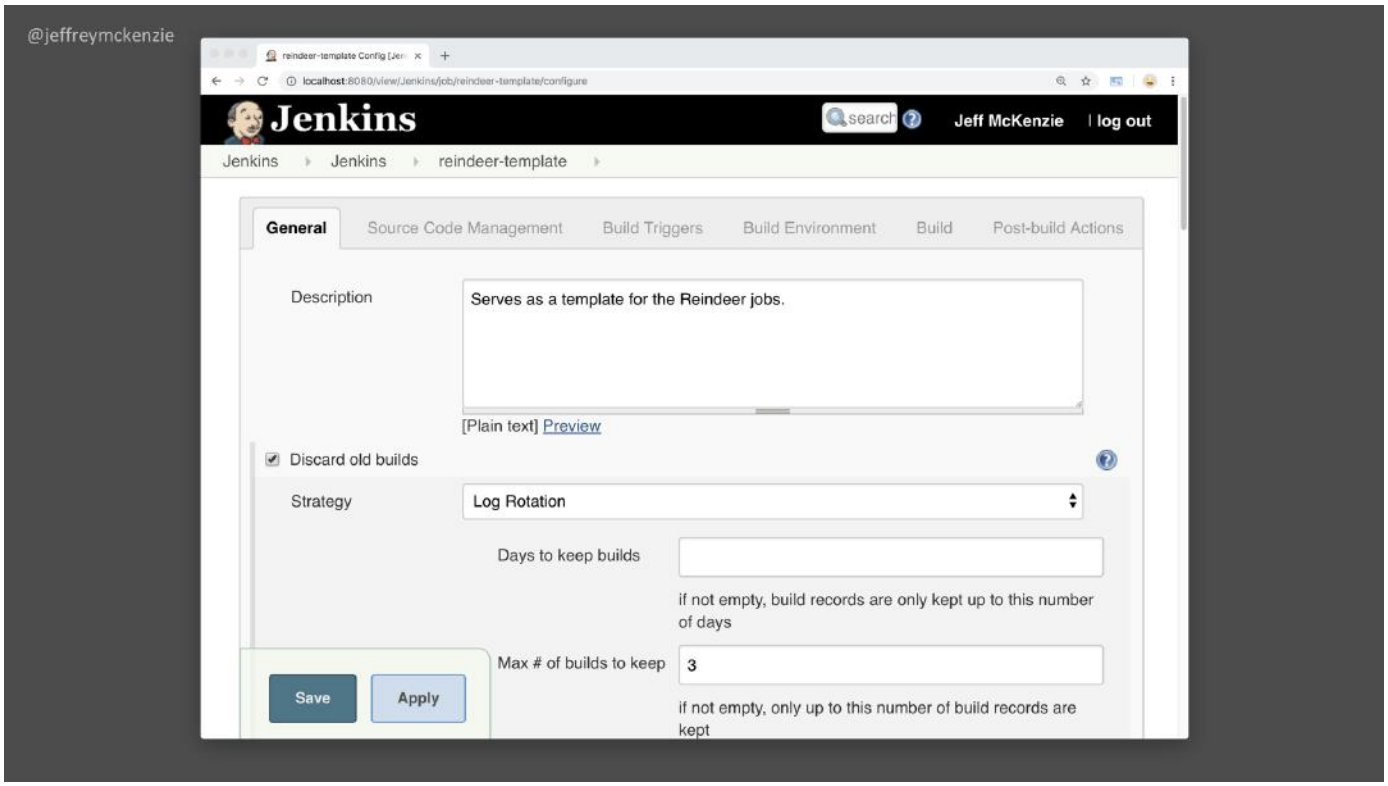


And we will create it as a copy of Rudolph

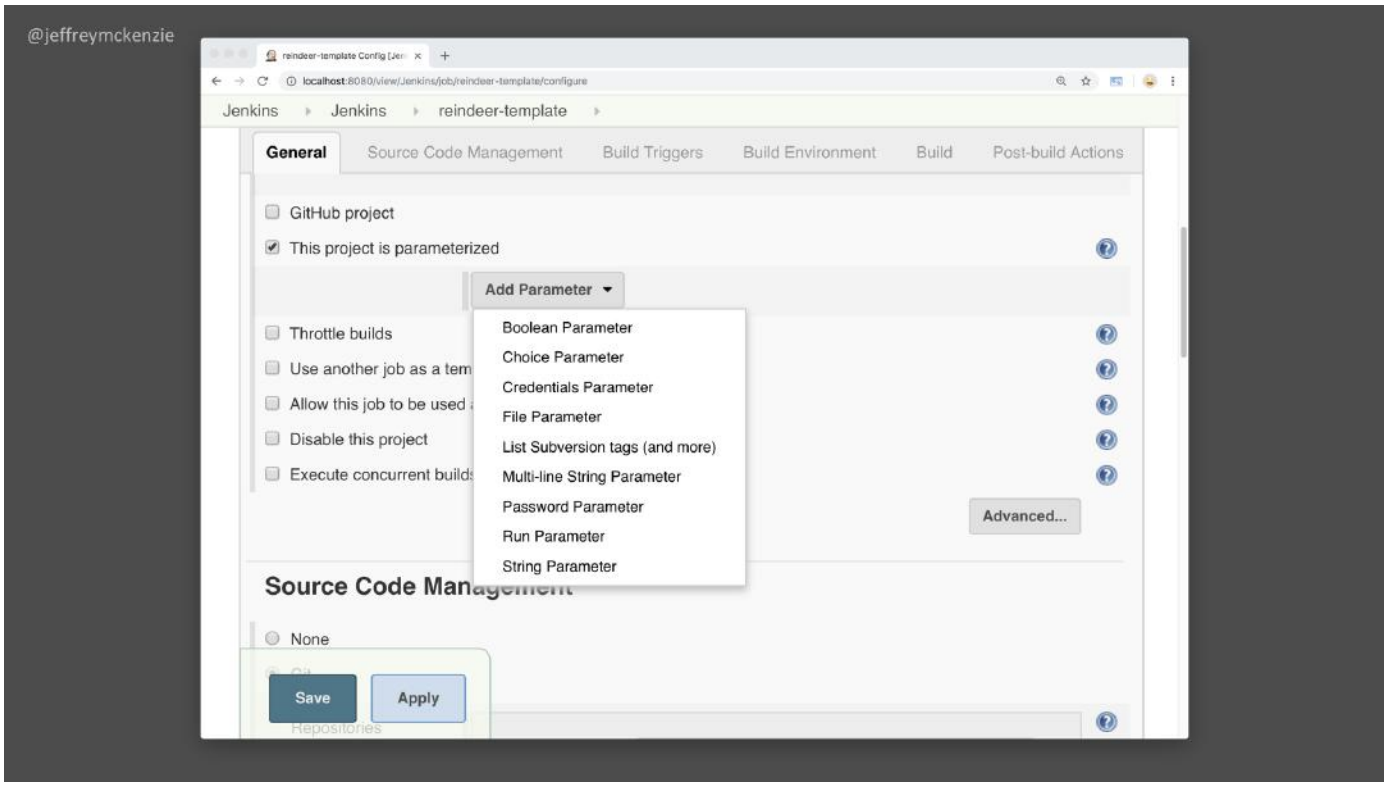
So we can use that as a starting point.



Let’s go ahead and change the description
So it makes sense as a template



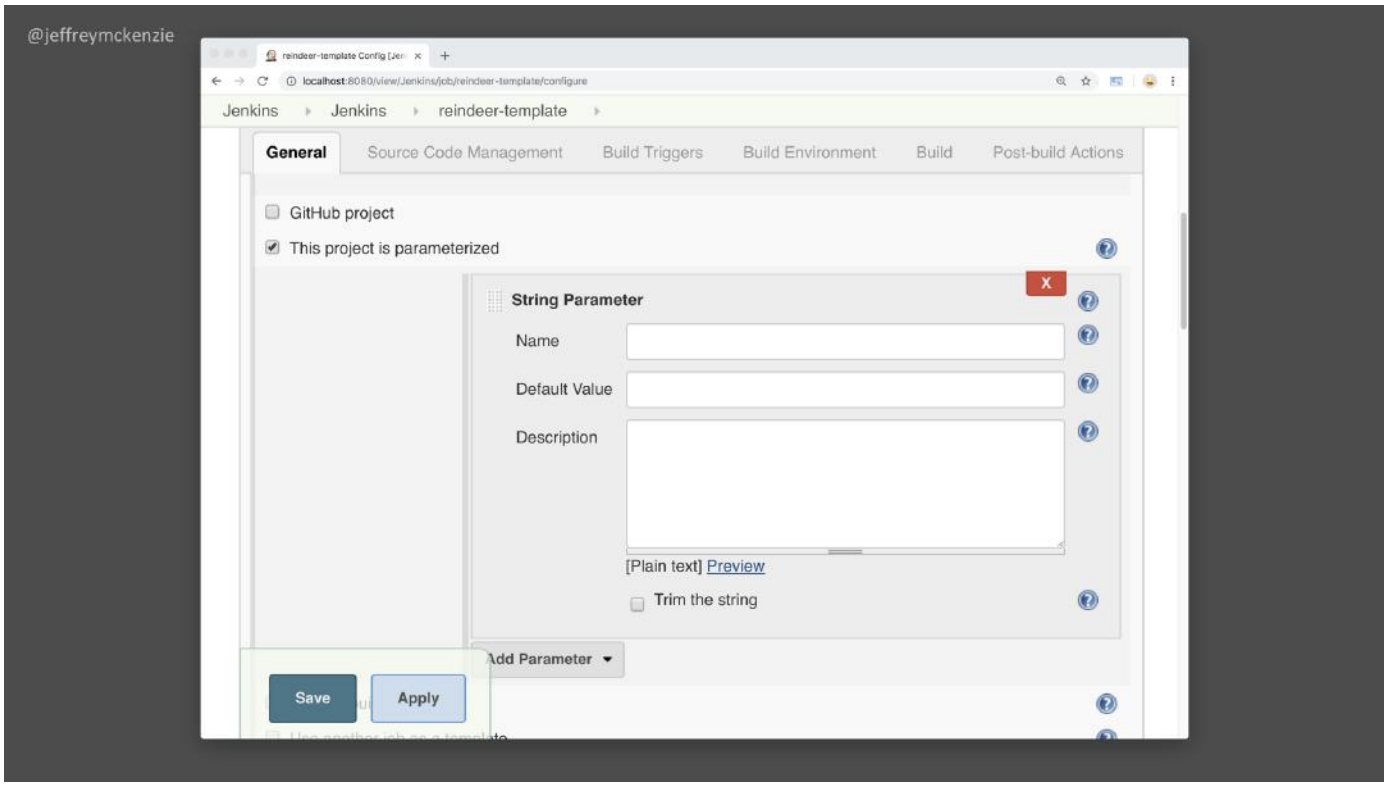
“Serves as a template for the reindeer jobs”



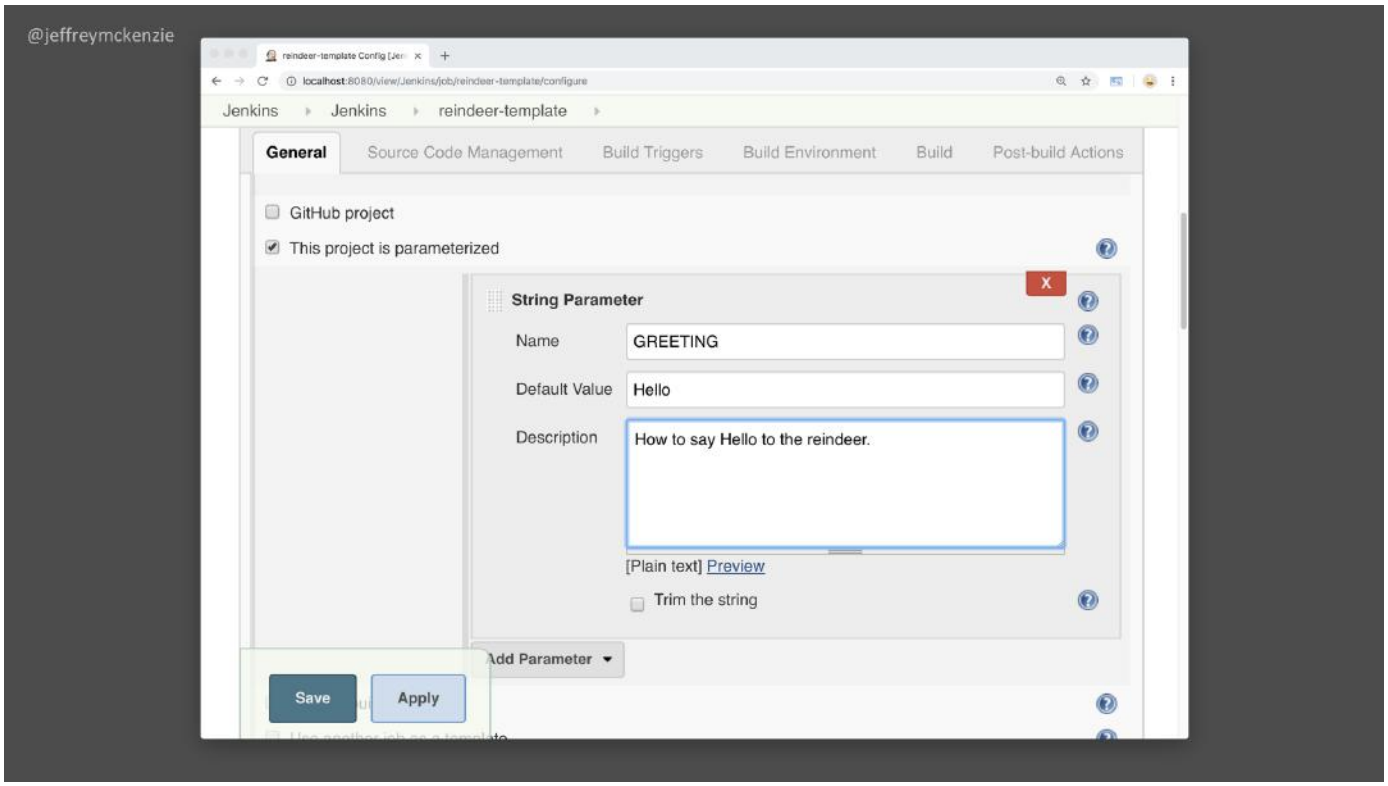
We are also going to add a couple of parameters (variables)
So Santa can personalize his reindeer greeting –

Check box for “project is parameterized”

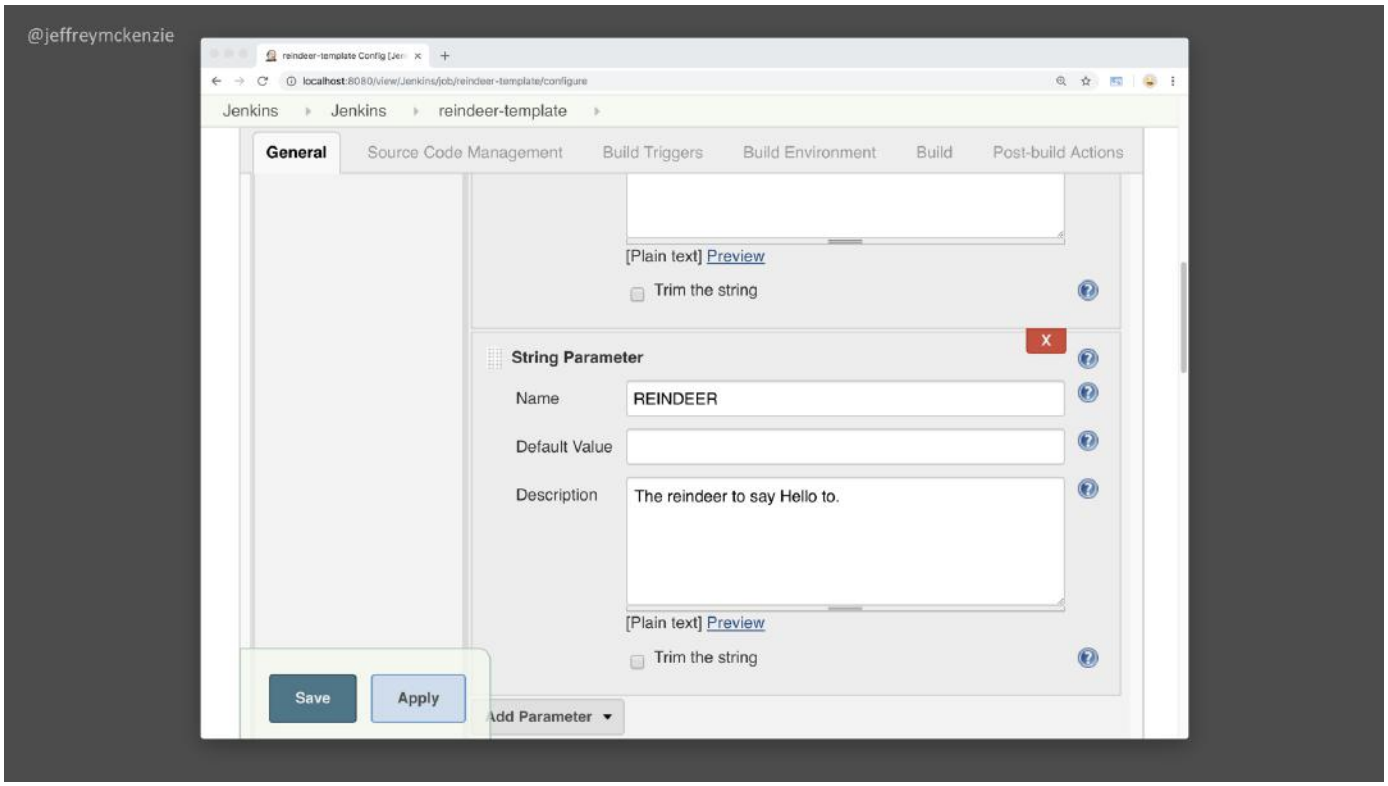
And we will select the string parameter



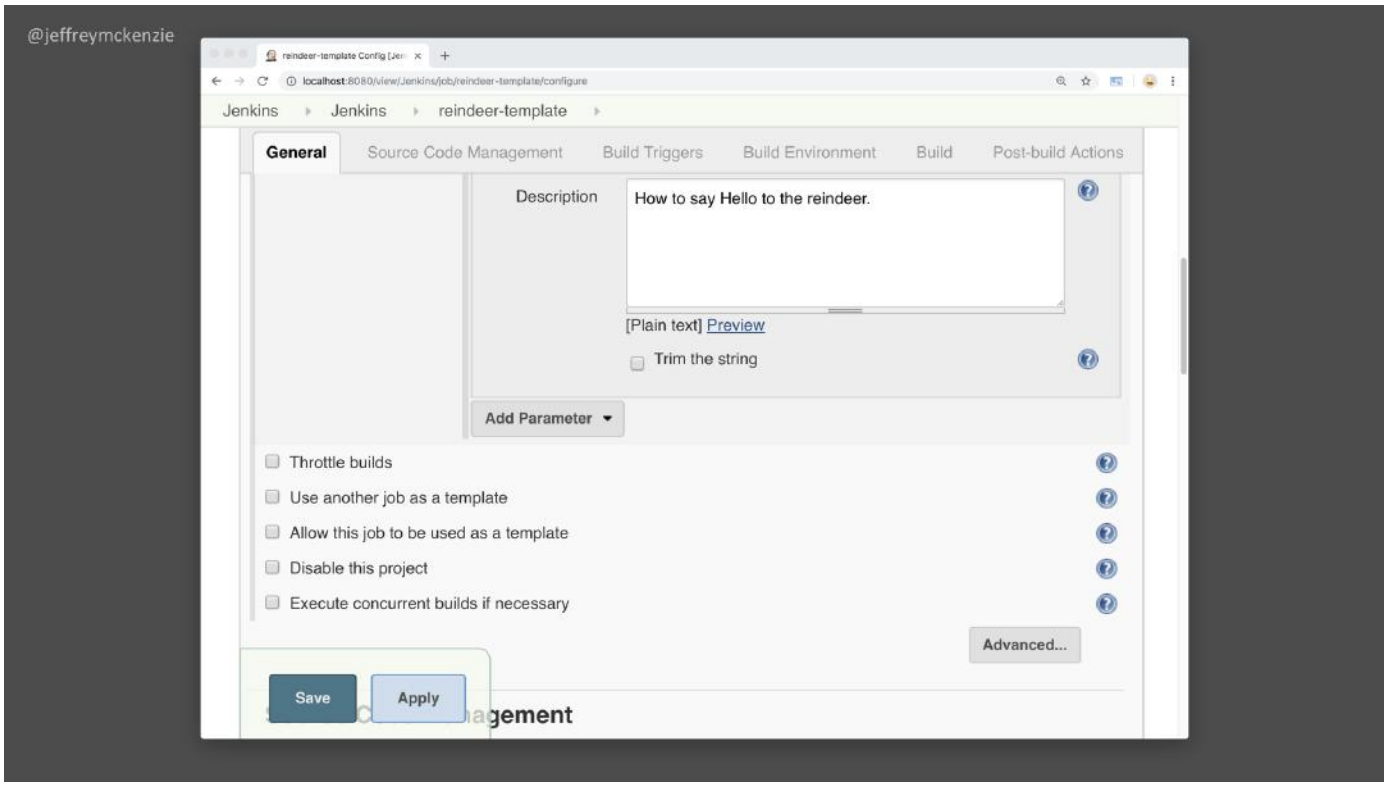
Let’s enter our values…



We'll call this parameter GREETING
With a default value of Hello
And also give it a description.

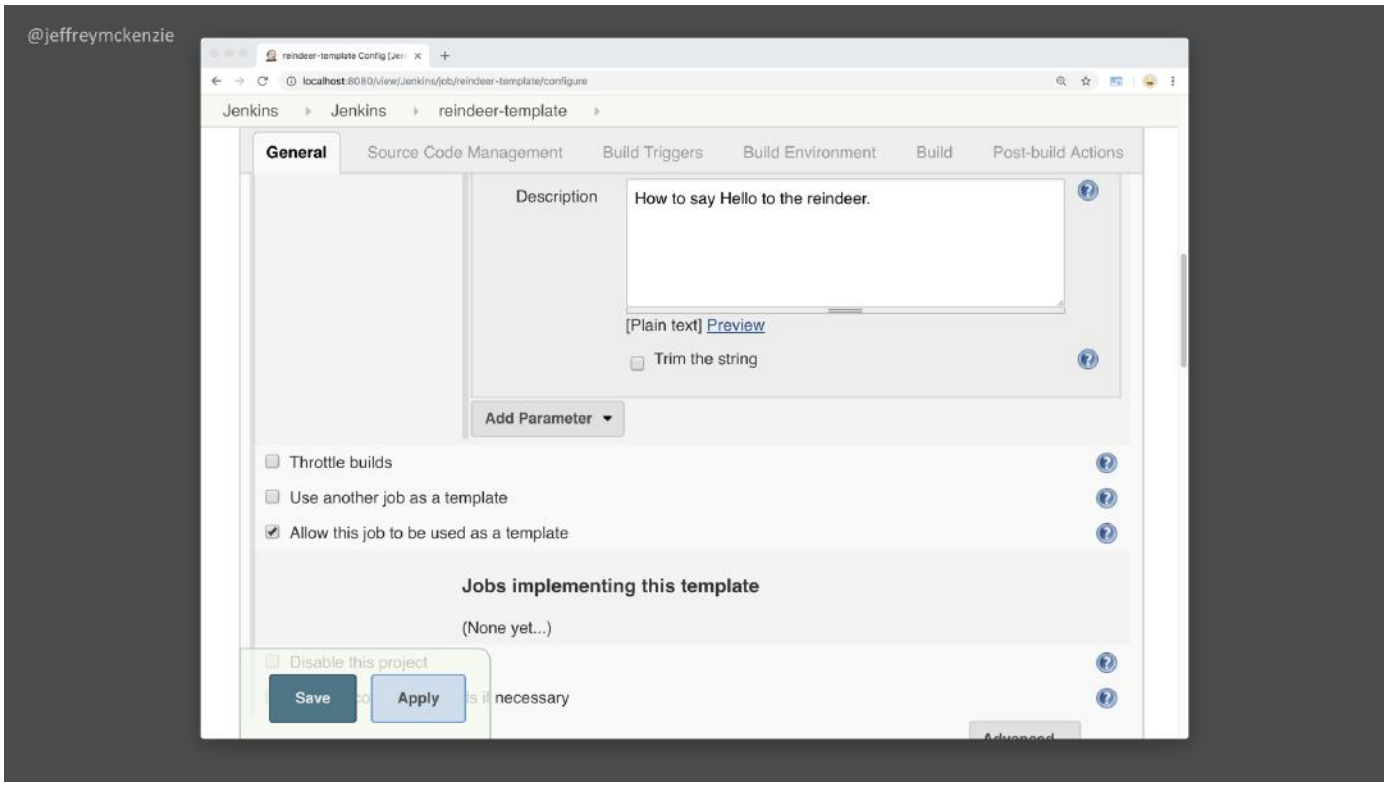


Next we will add parameter REINDEER
With no default value
And also give it a description –
“The reindeer to say hello to”

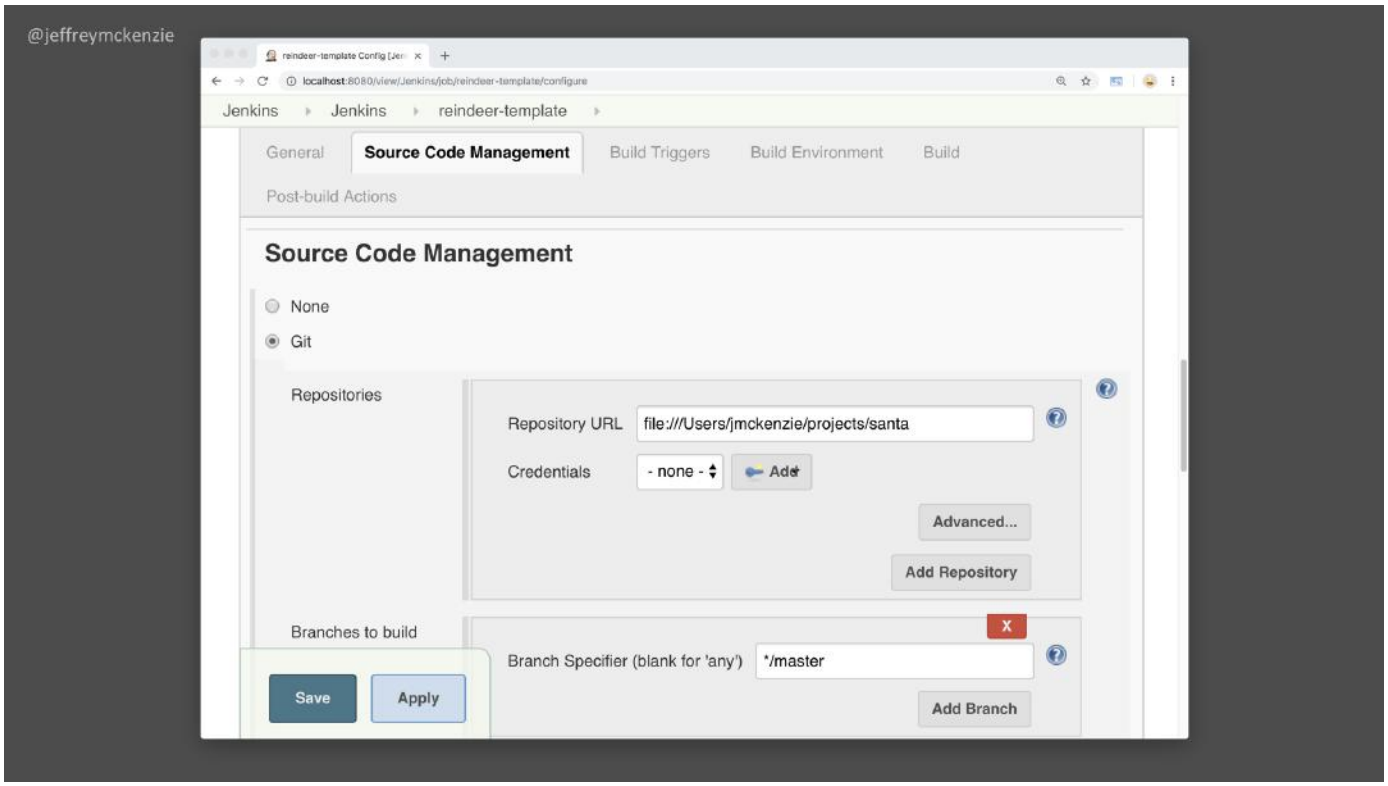


Next we will scroll down to the checkbox that says
“Allow this job to be used as a template”

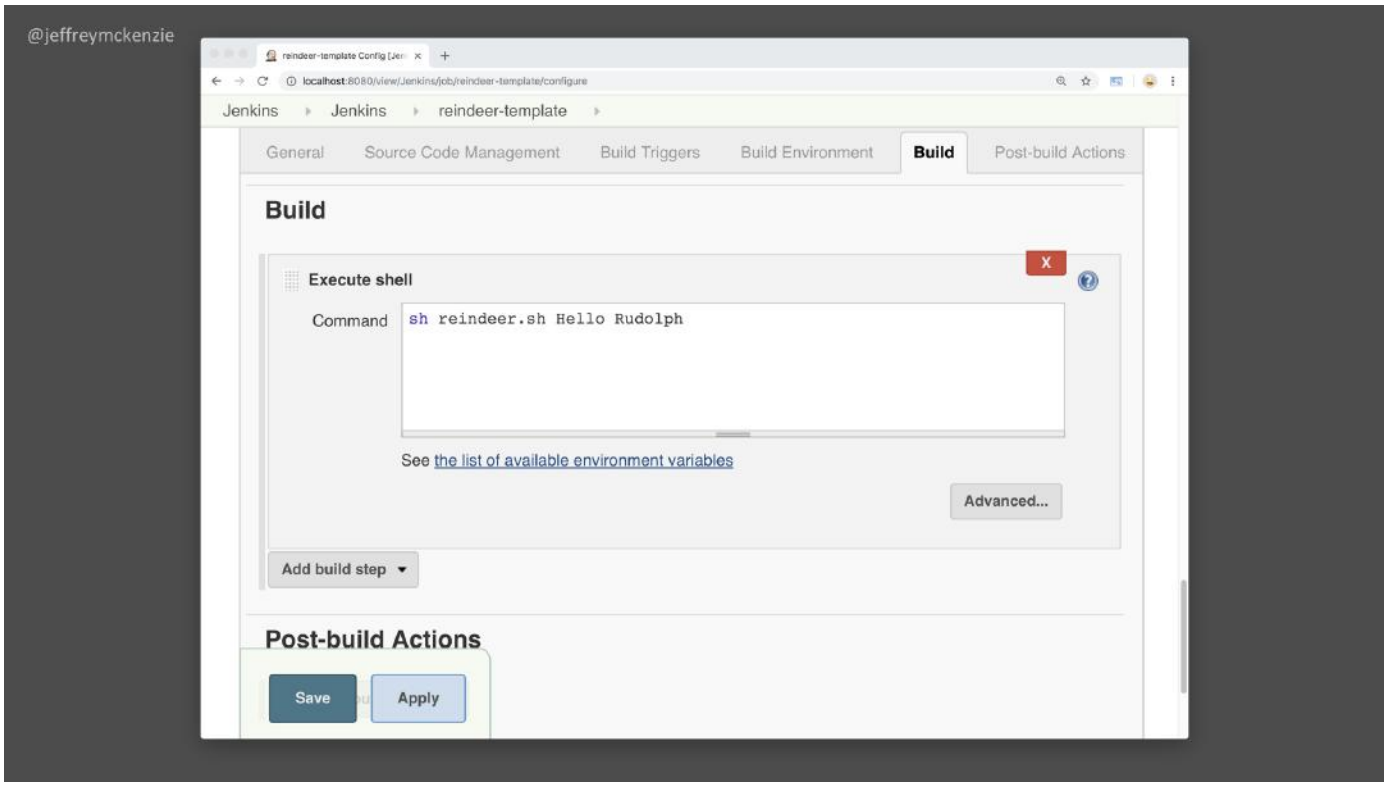
[you see this option only if you have EZ Templates installed]



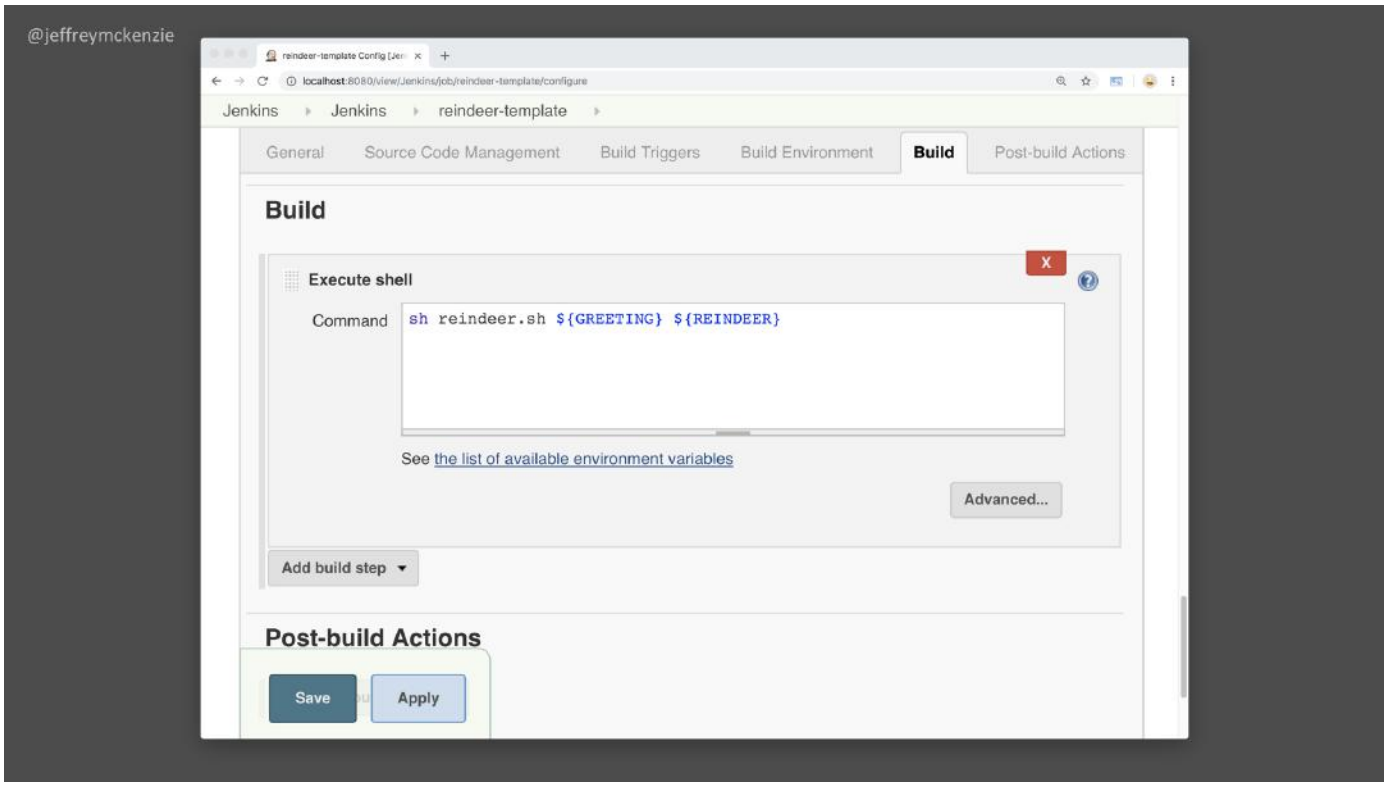
so we check that...



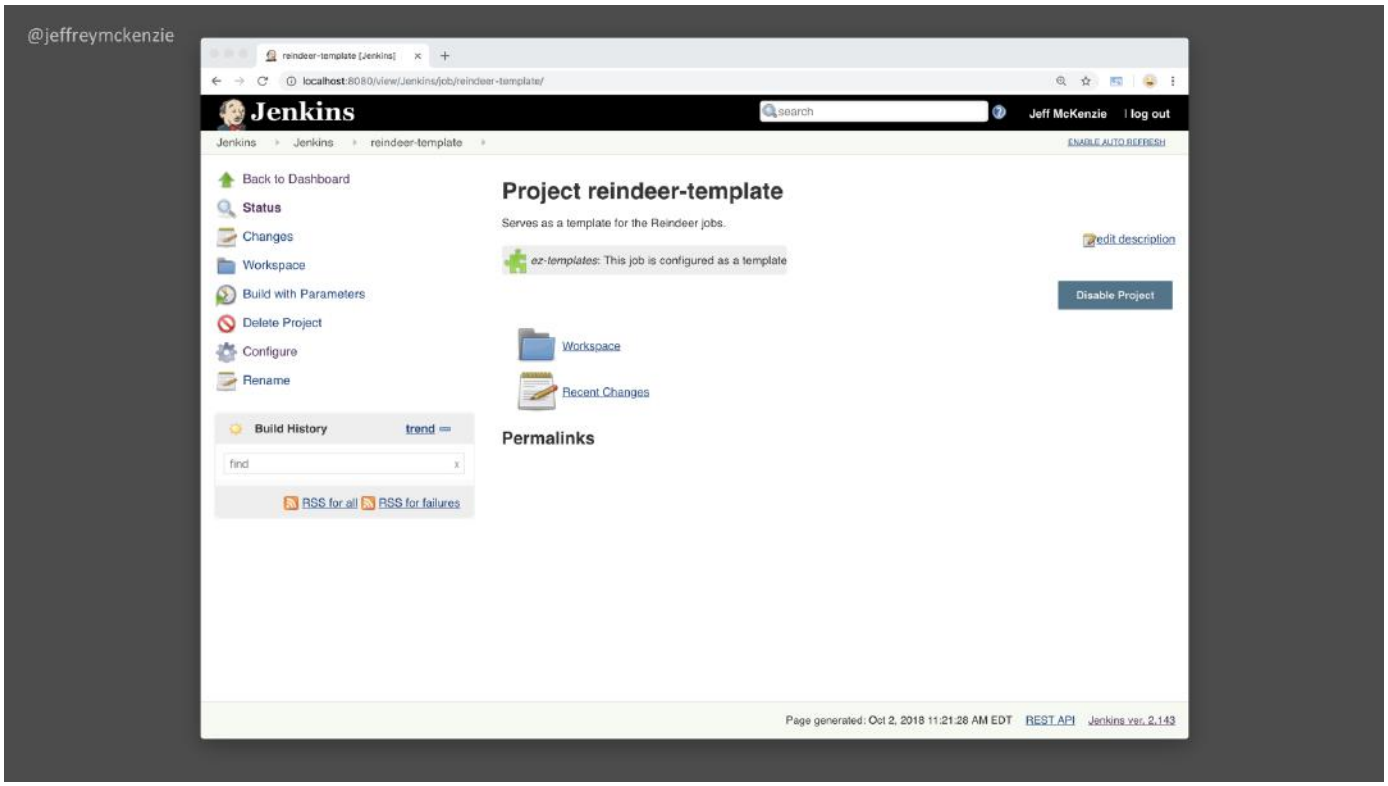
We leave our git repo as it is….



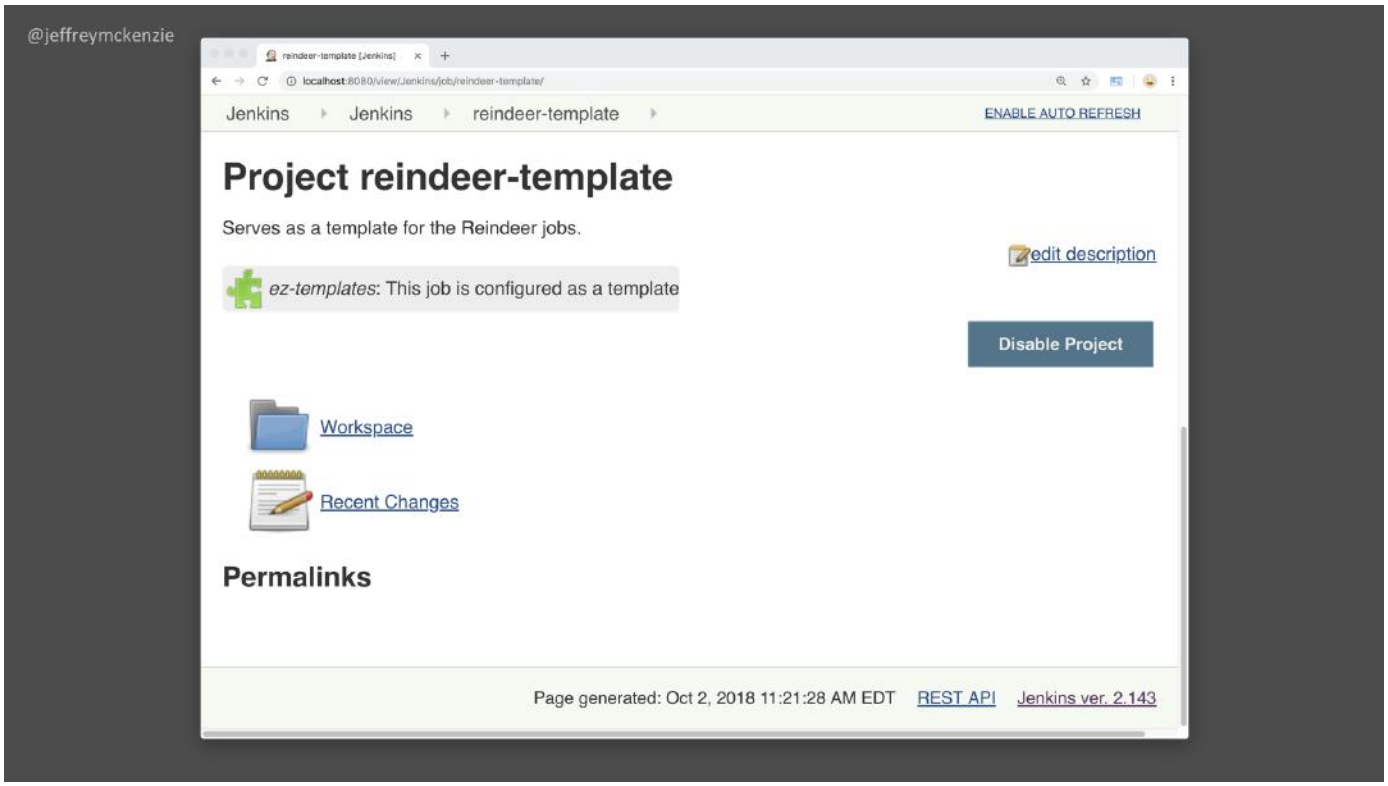
We will also change our shell command
To use the parameters



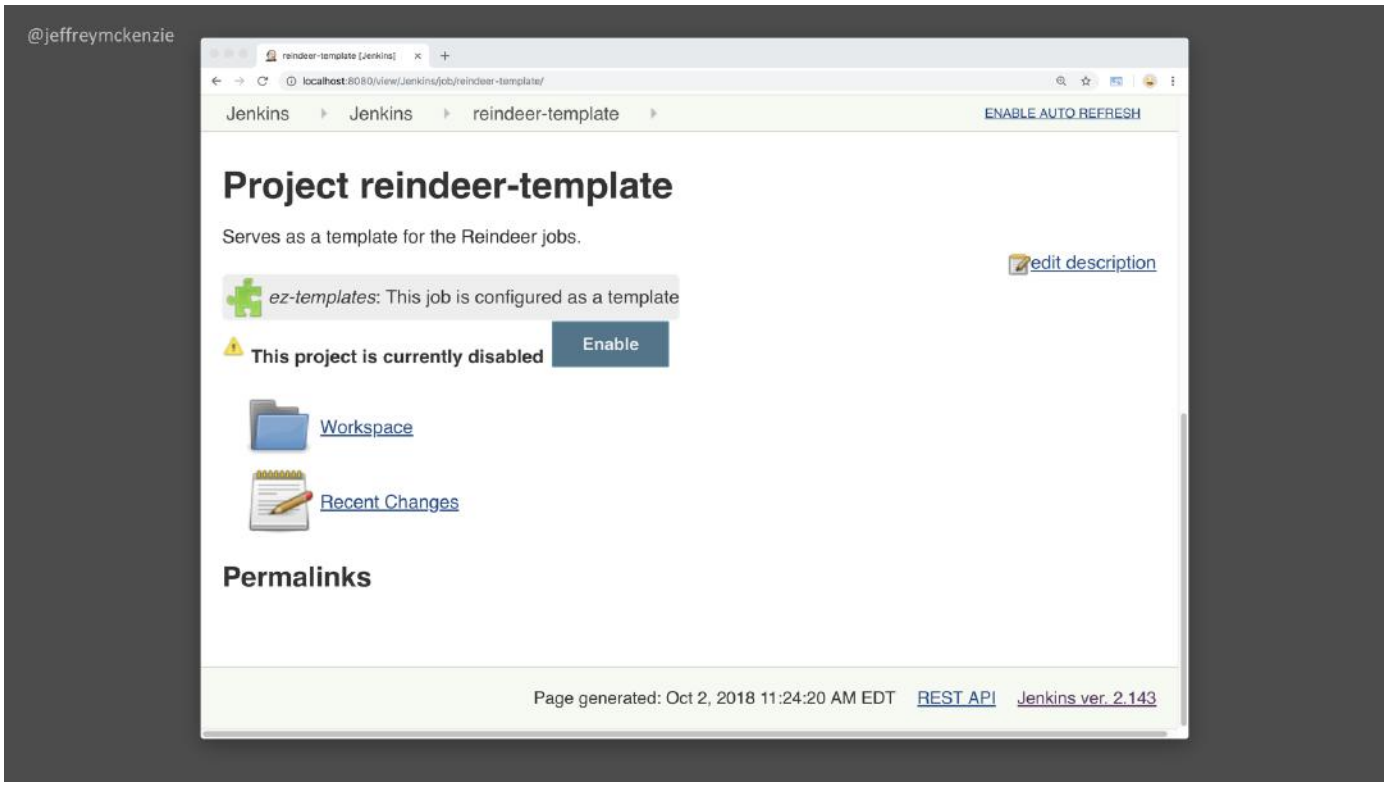
Then we will click save…



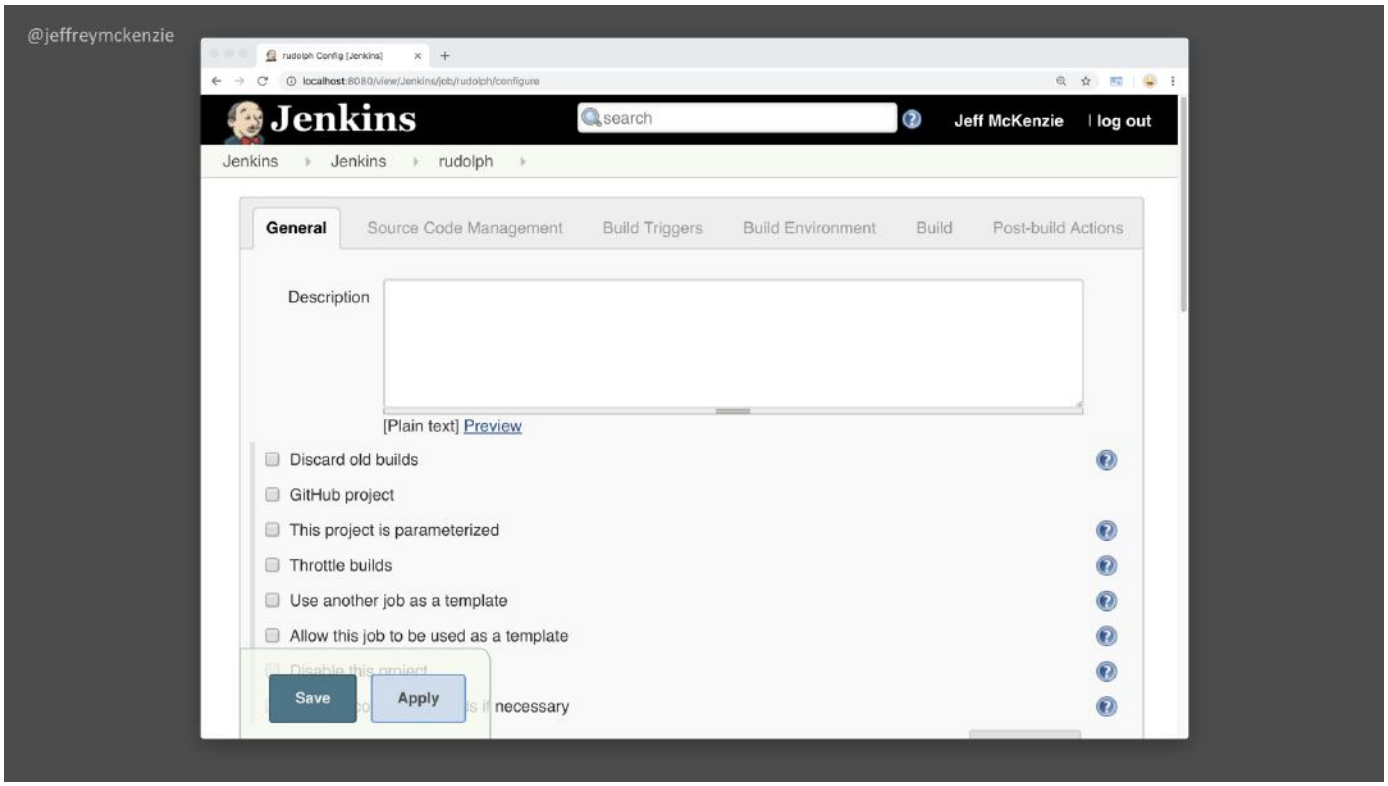
Which takes us back to the project page



Because this is a template and not something we want to run,
We will disable the project, which is important
If the template contains build triggers
Or anything like that.

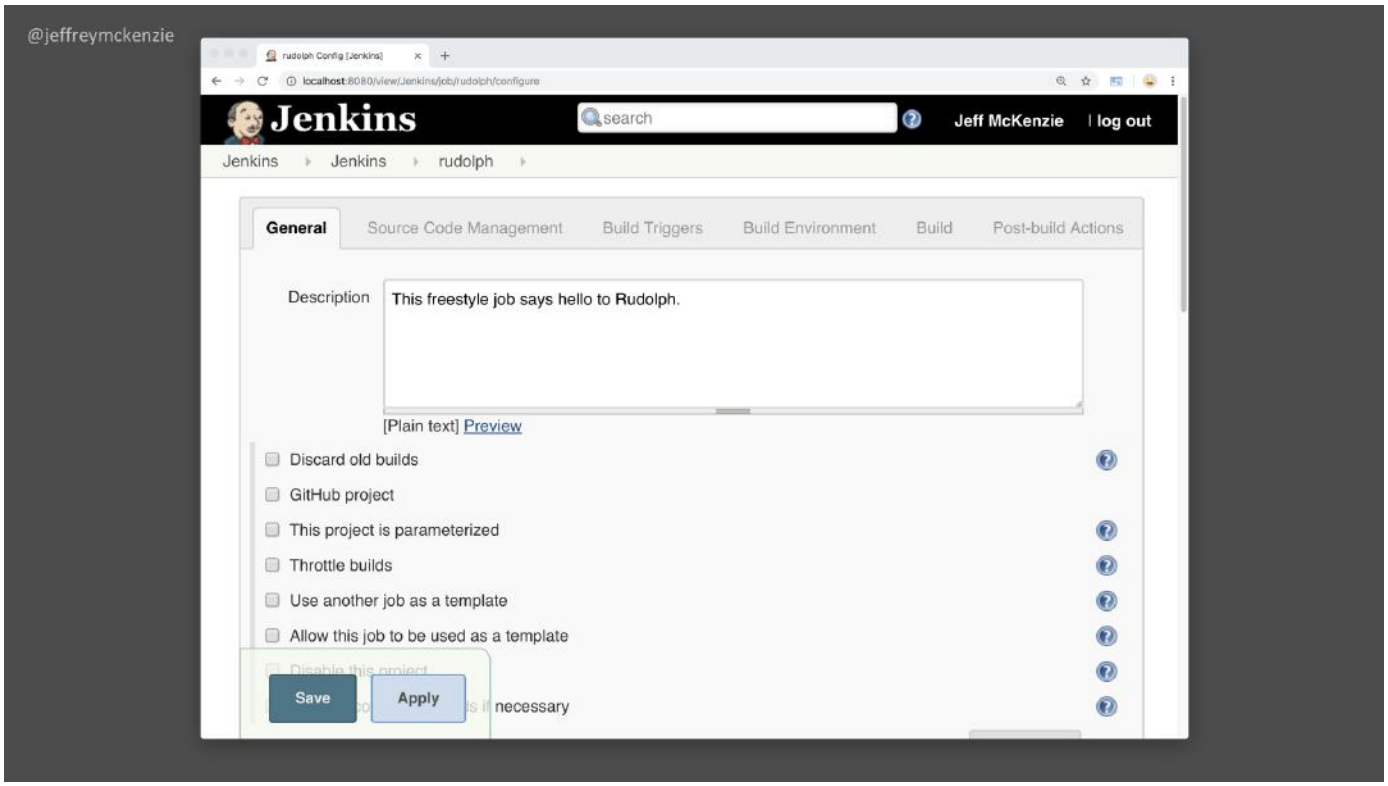


So we will do that
And it shows up as disabled.



Let’s re-create our rudolph freestyle job
To see how to create a child job

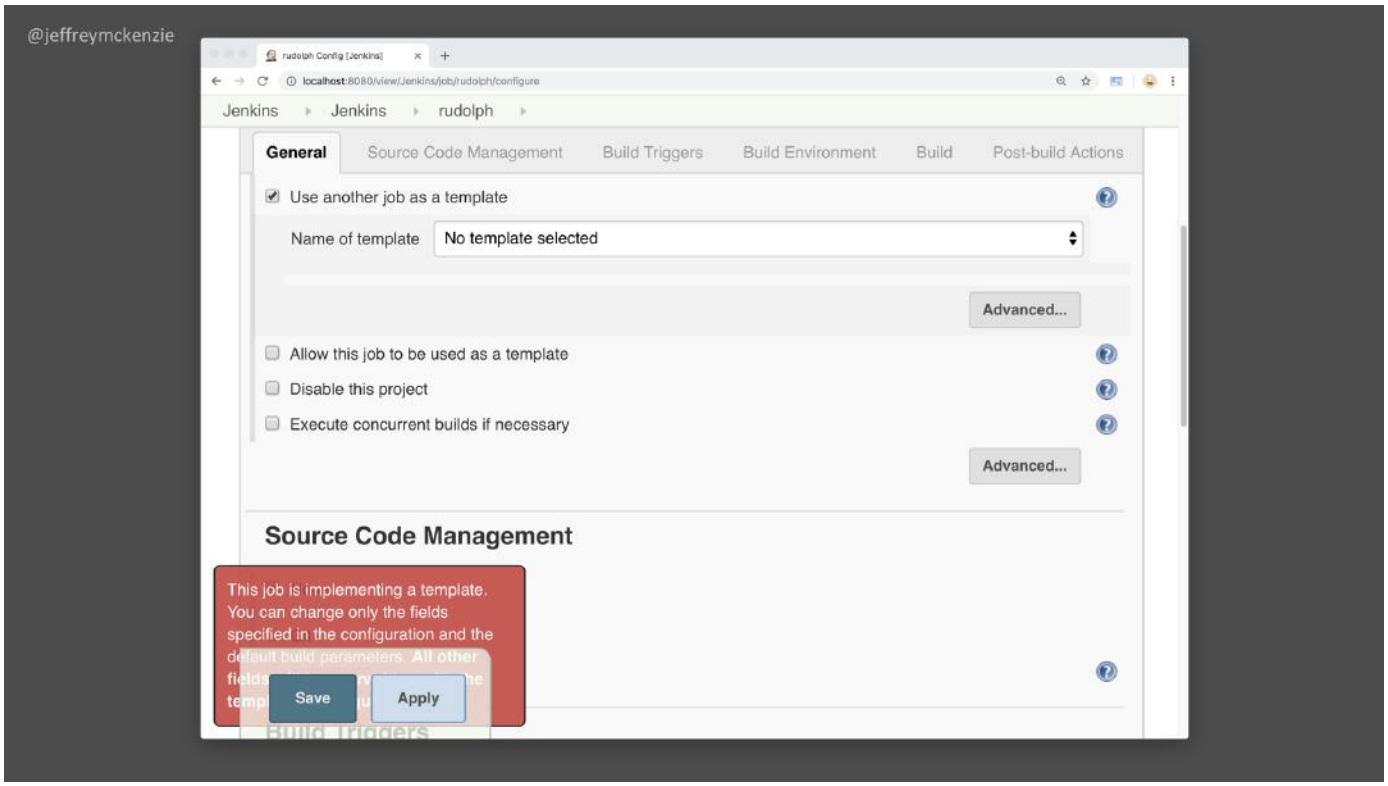
We’ll use the same description we had before….



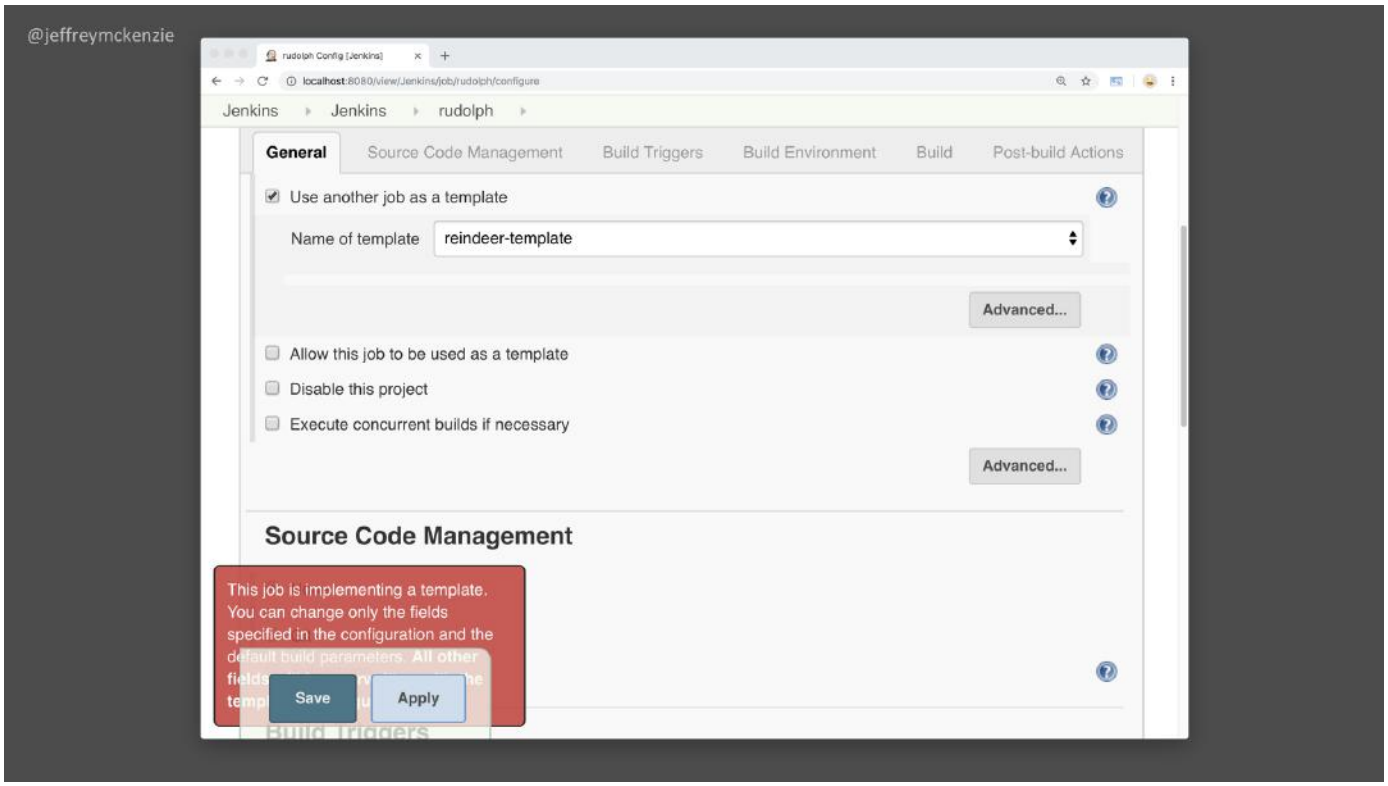
This freestyle job says hello to Rudolph….

Then we scroll down to the checkbox that says

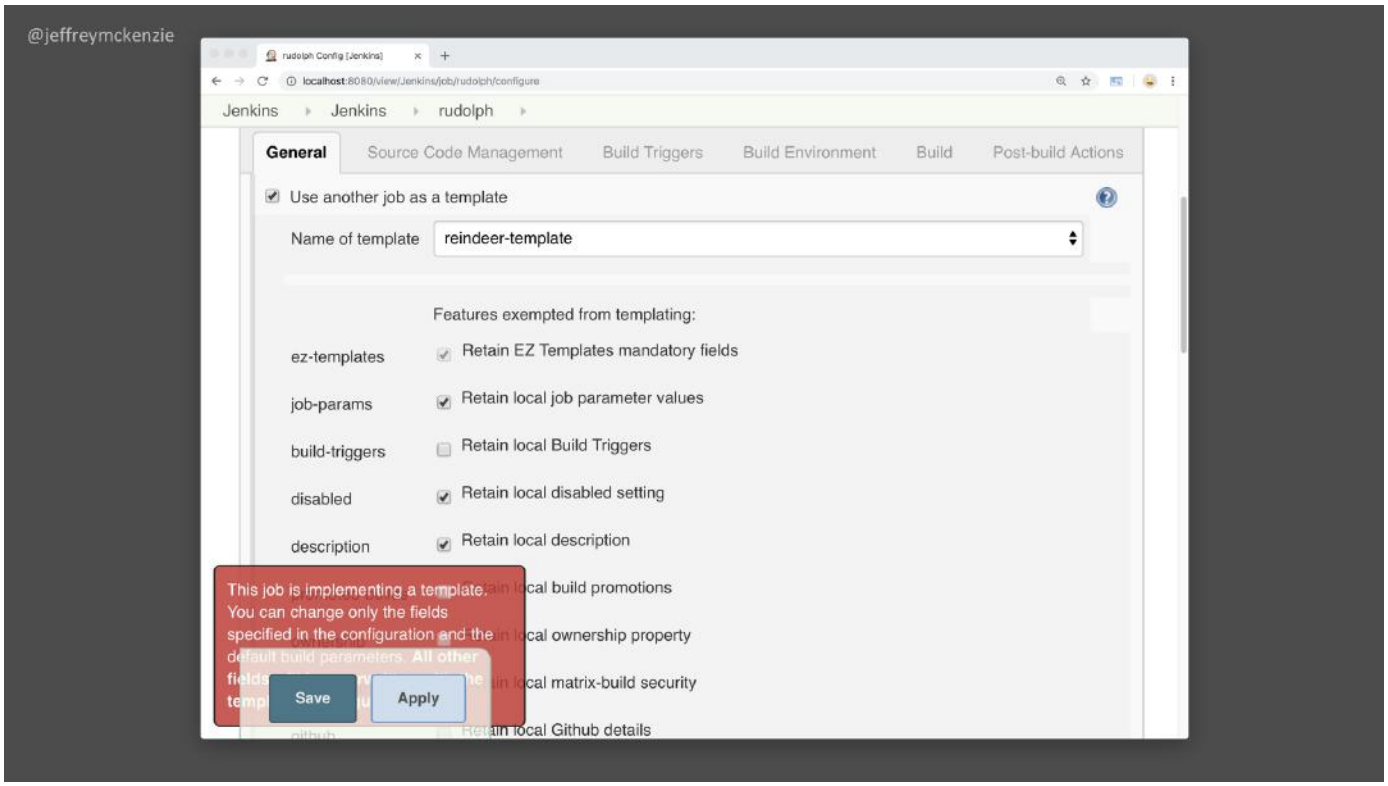
“use another job as a template”



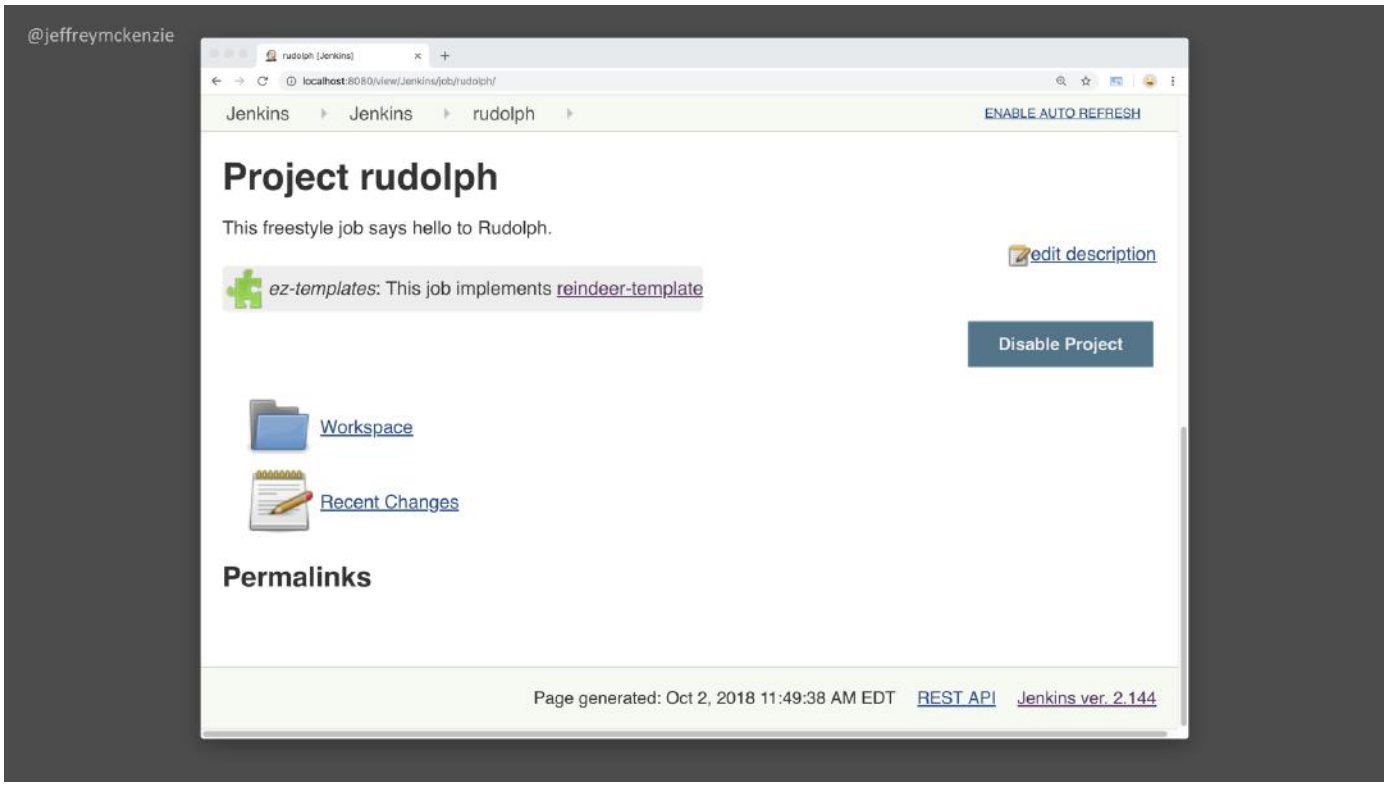
When we check that we are given a dropdown
Of what template we want to select



We will choose reindeer template



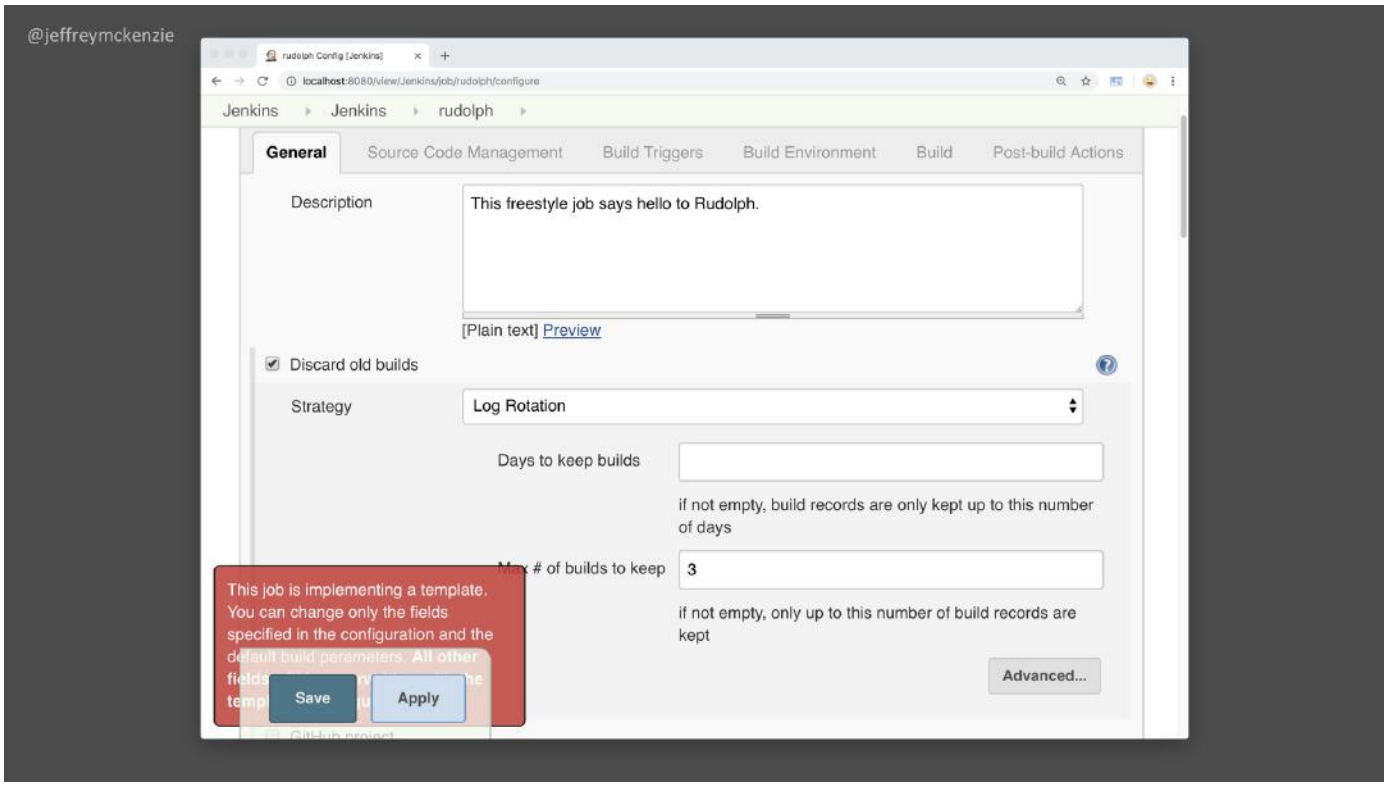
so Let’s save that and go back to the project page



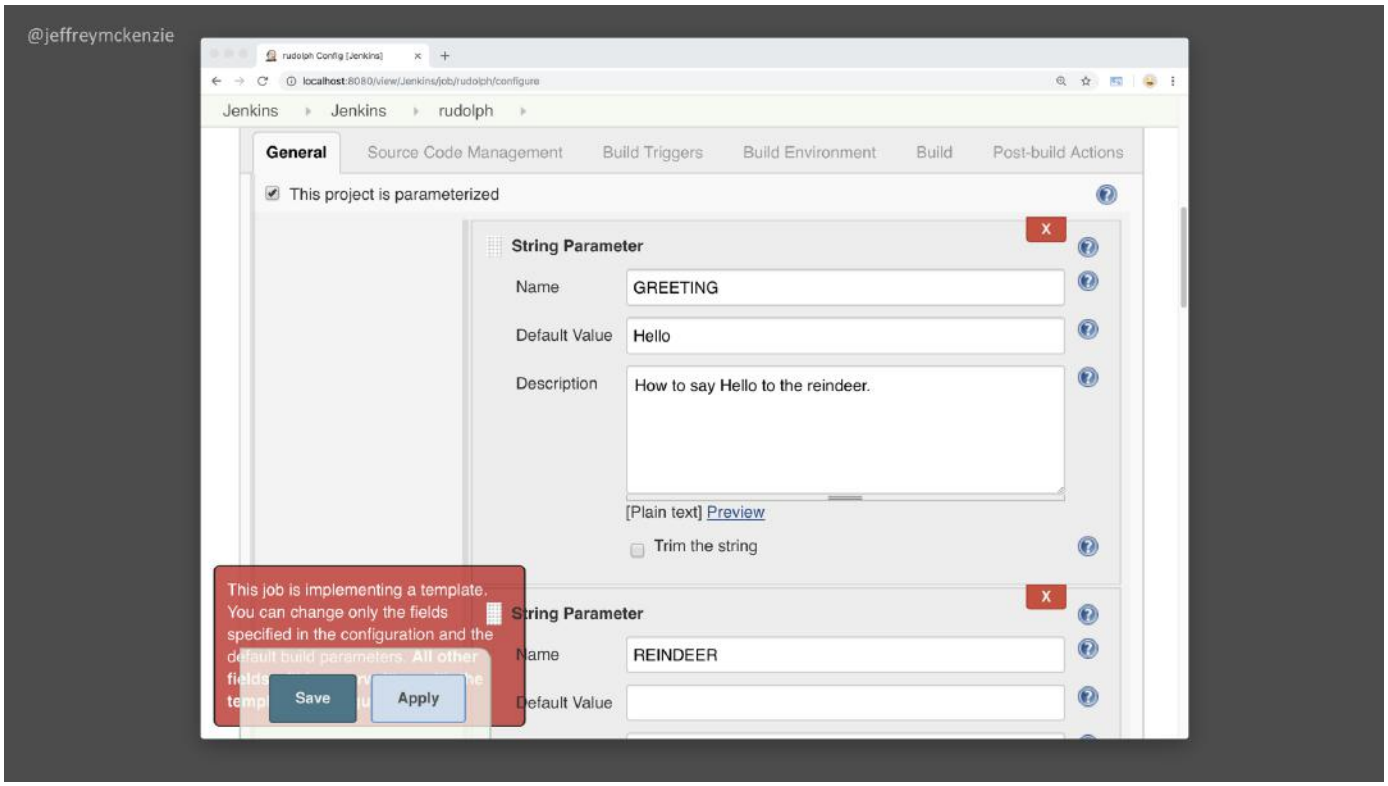
This shows us that Rudolph
Is implementing the reindeer template.

Remember, all we did was add a description,
Select a template, and save it.

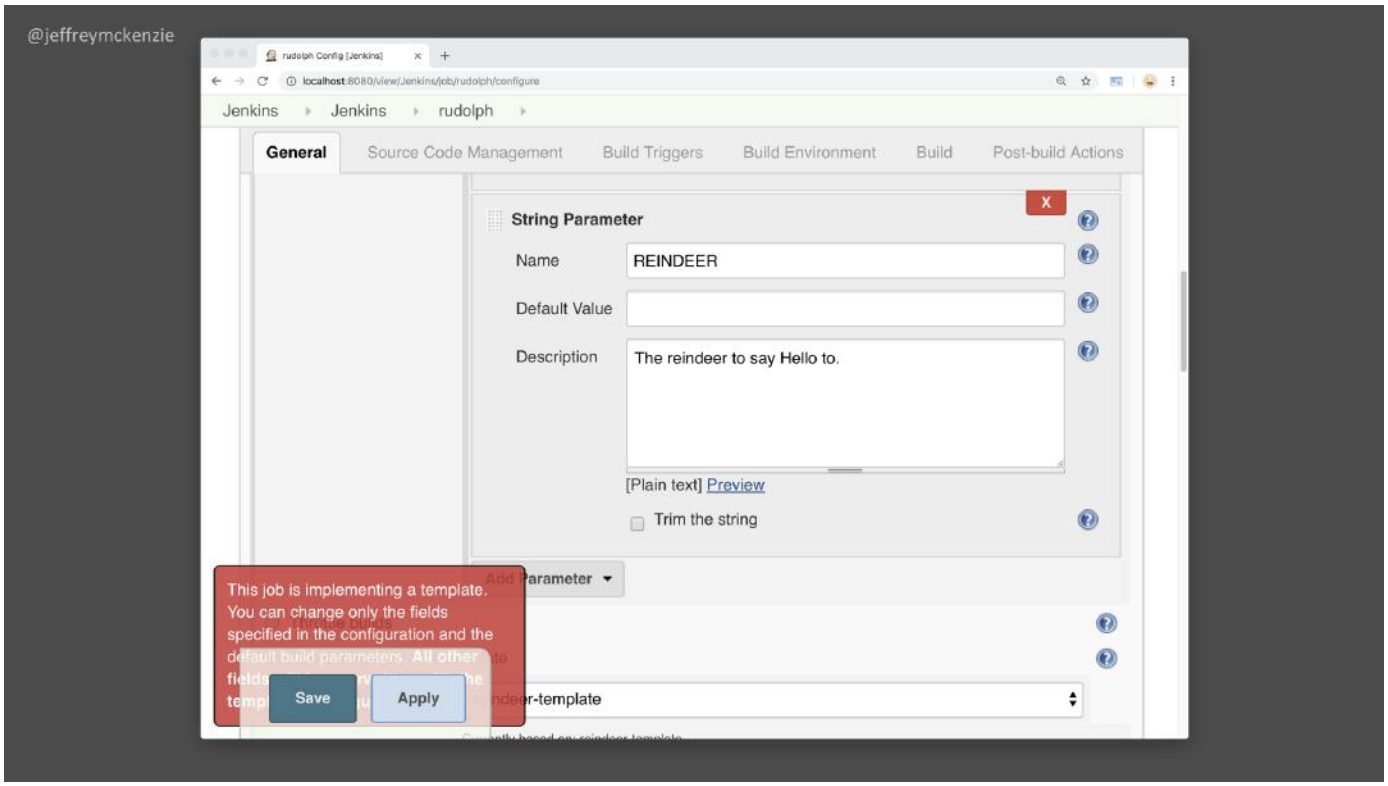
Let’s go back into the configuration.



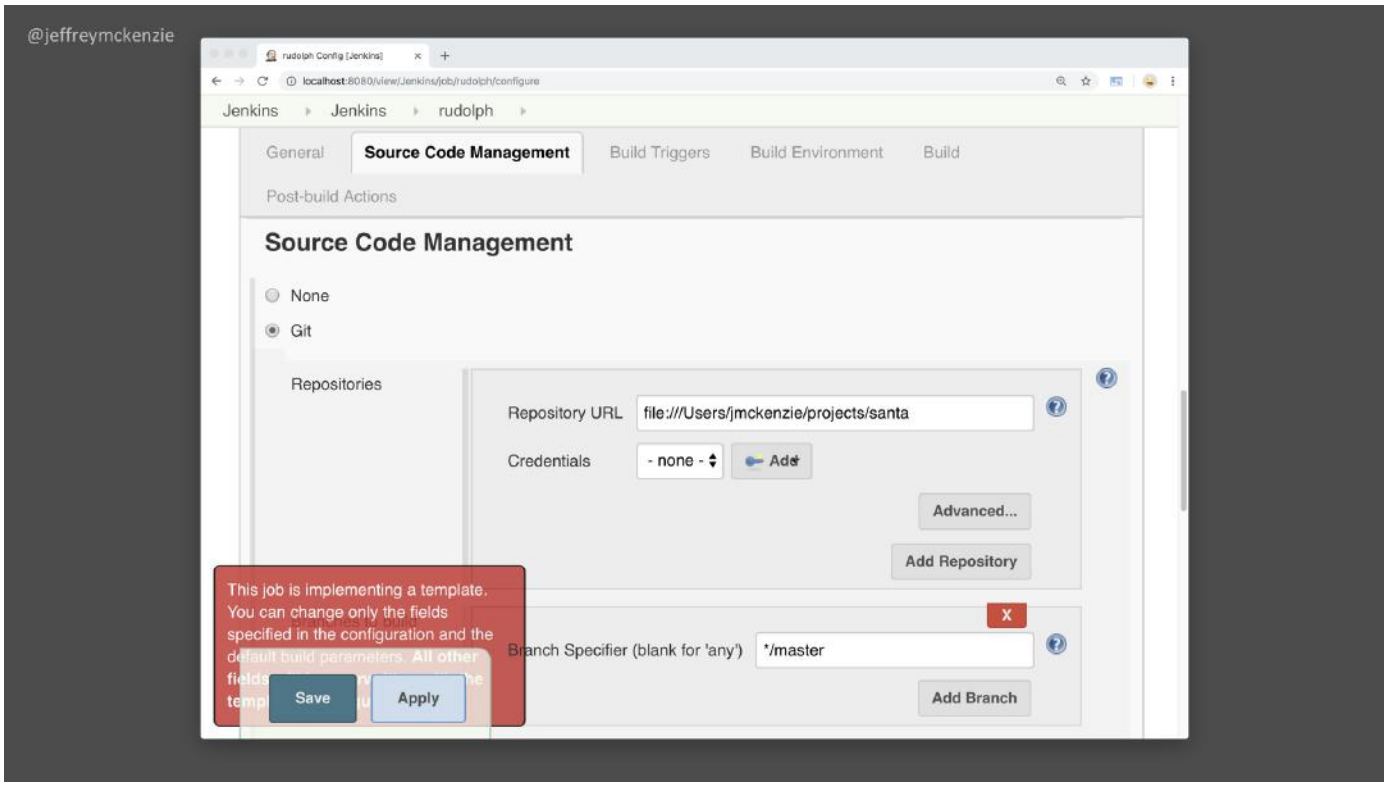
We have our description,
Our build history setting…



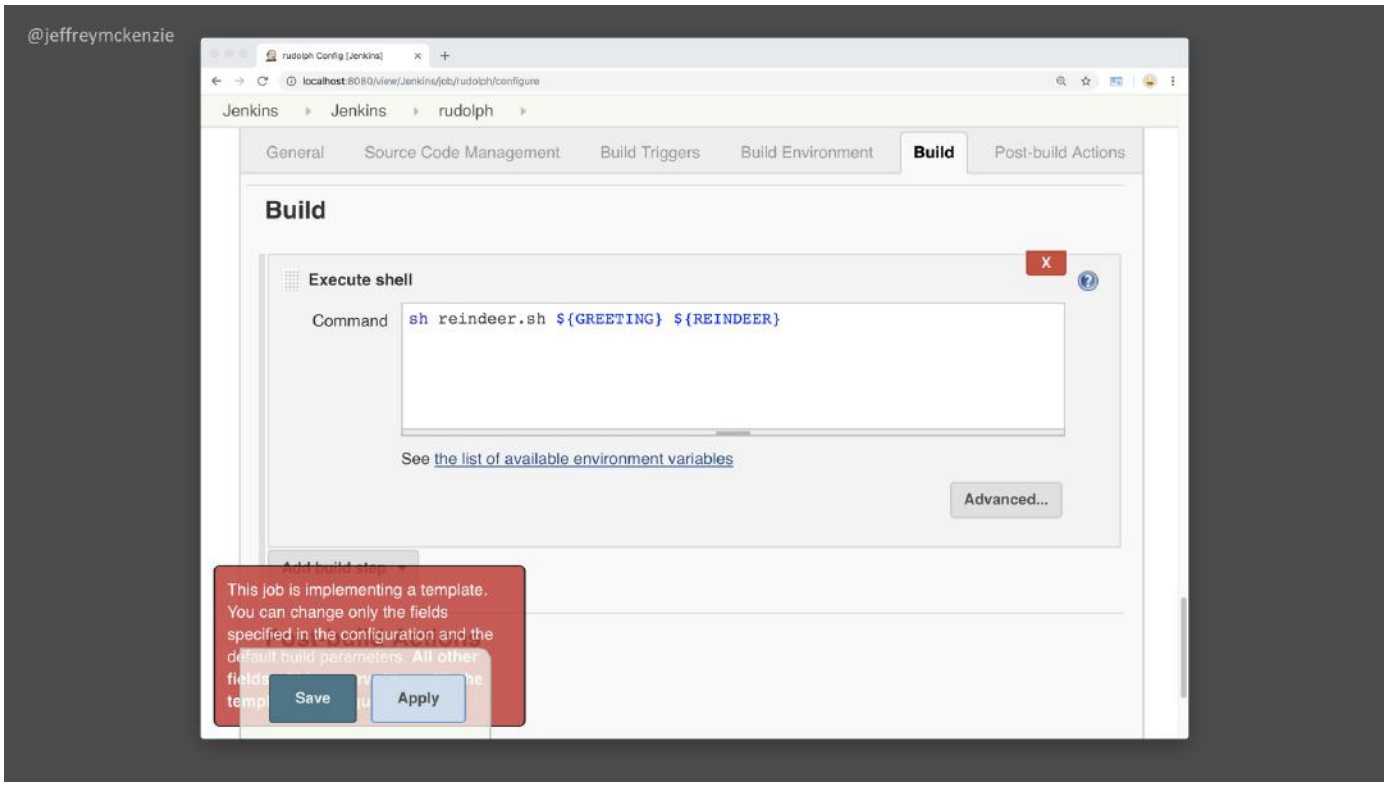
Our GREETING parameter….



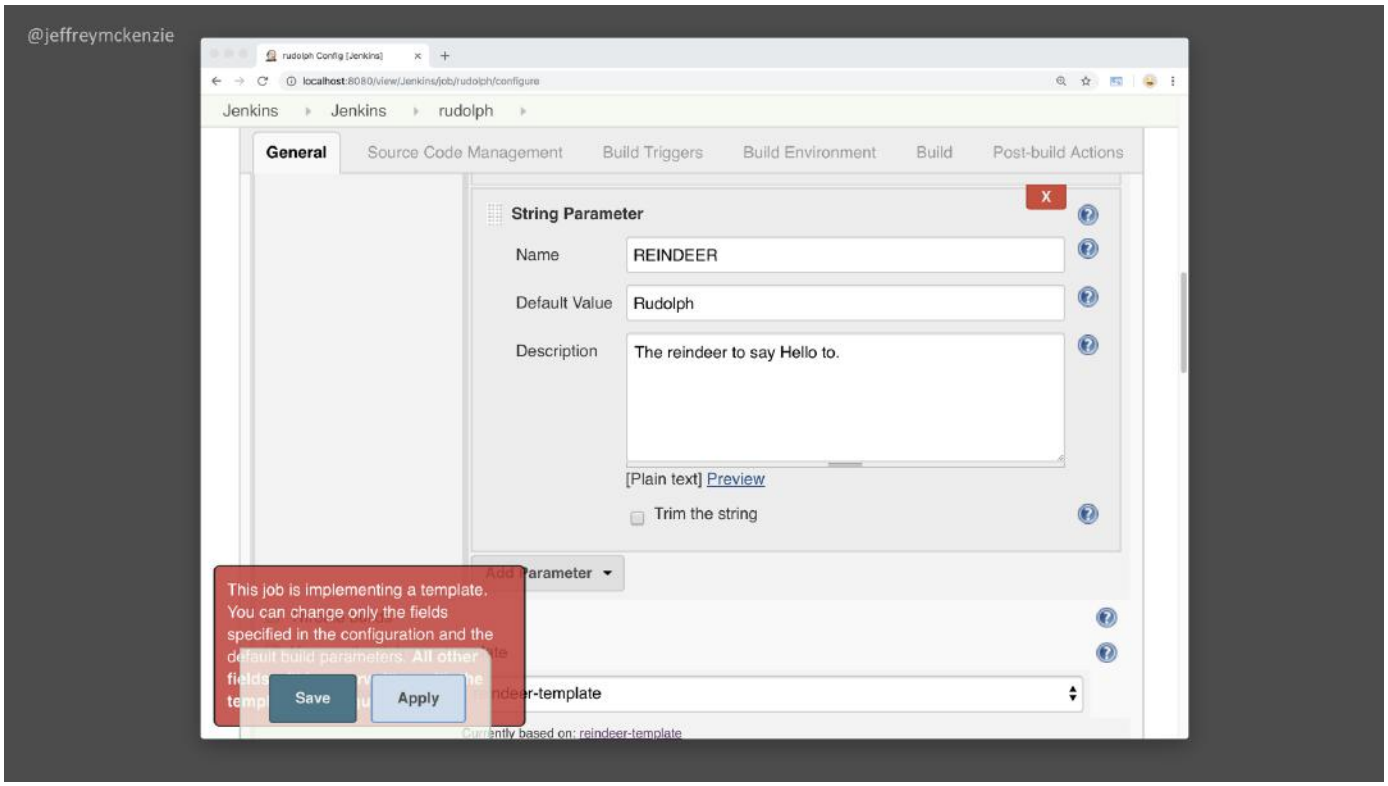
Our REINDEER parameter



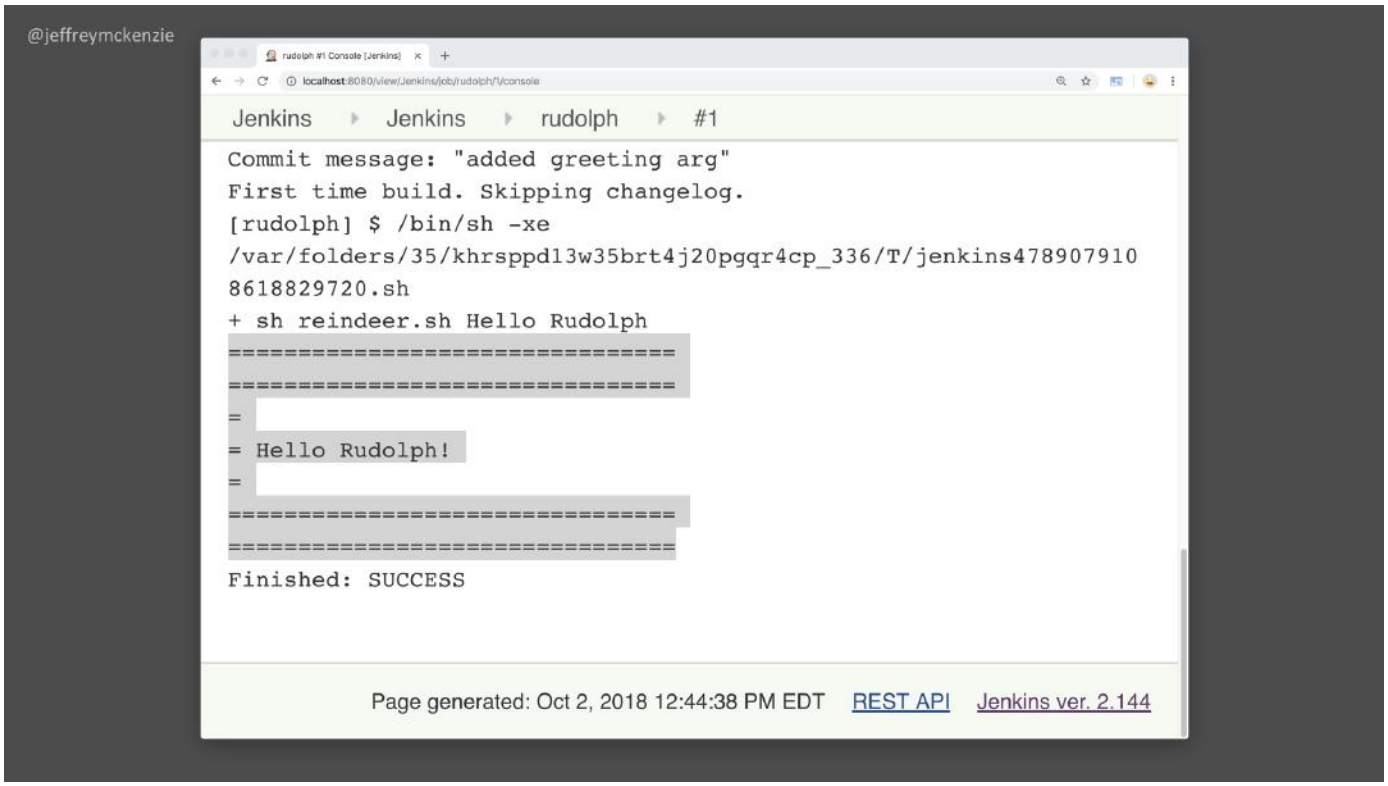
Our git settings…



And our build step.



So the only thing we have to do is
fill in the reindeer parameter
With Rudolph,
And run the project.



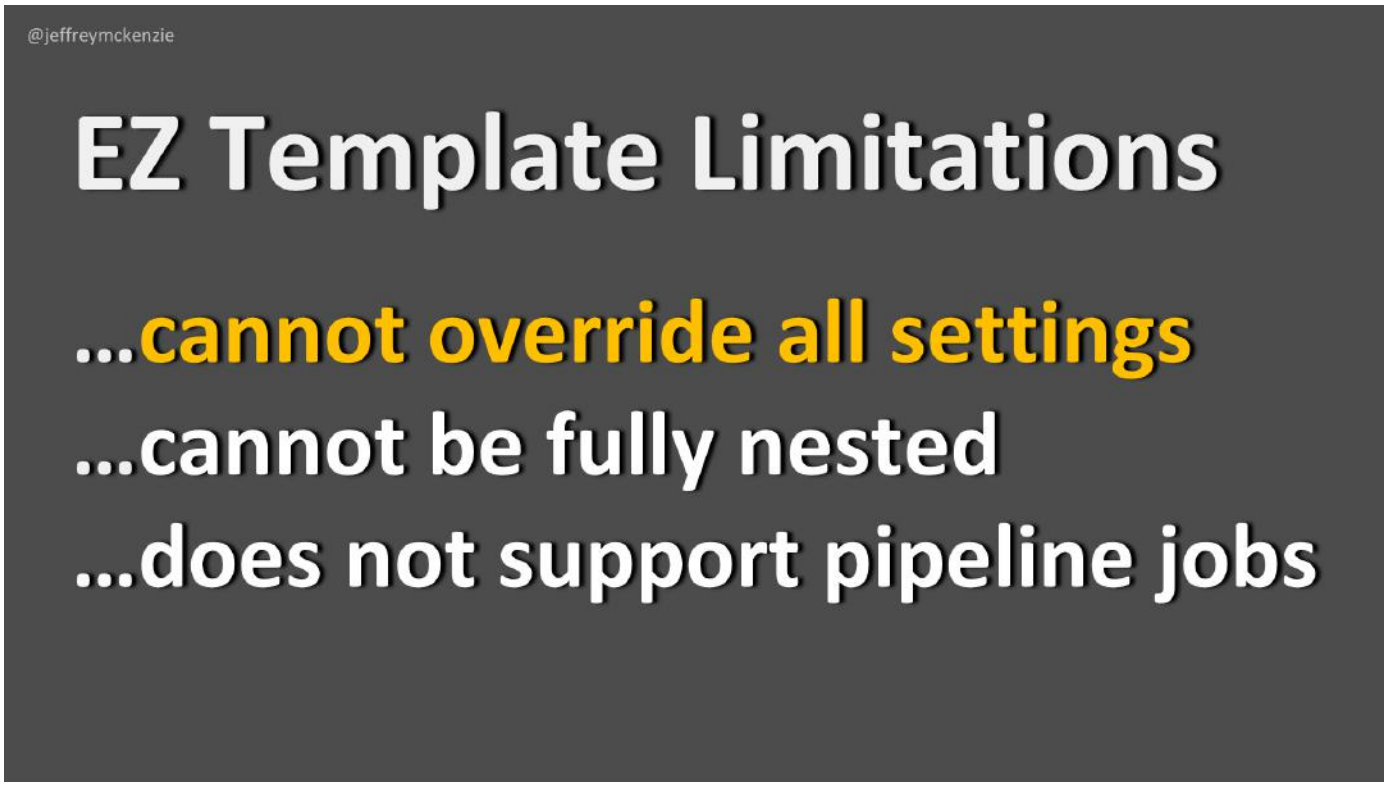
And our result is the same as before.

So now Santa can talk to as many reindeer as he wants,

And since they are all based on the same template,

The projects are much easier to change and maintain.

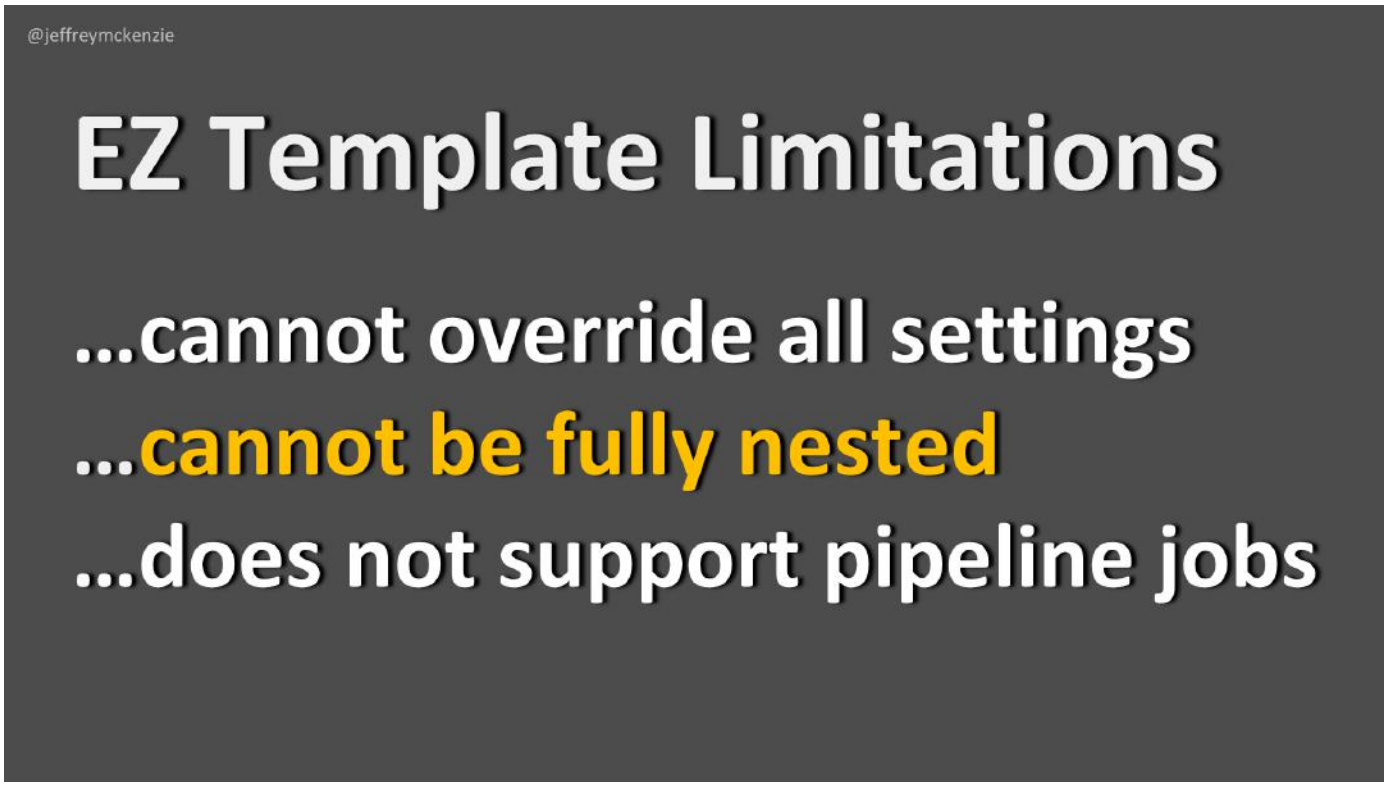
Those are the benefits, but there are limitations to this approach.



First is not all settings can be overridden.

For example, if we wanted to modify

Our build step, our shell command, can only do so
In the template – local changes to that don't take effect

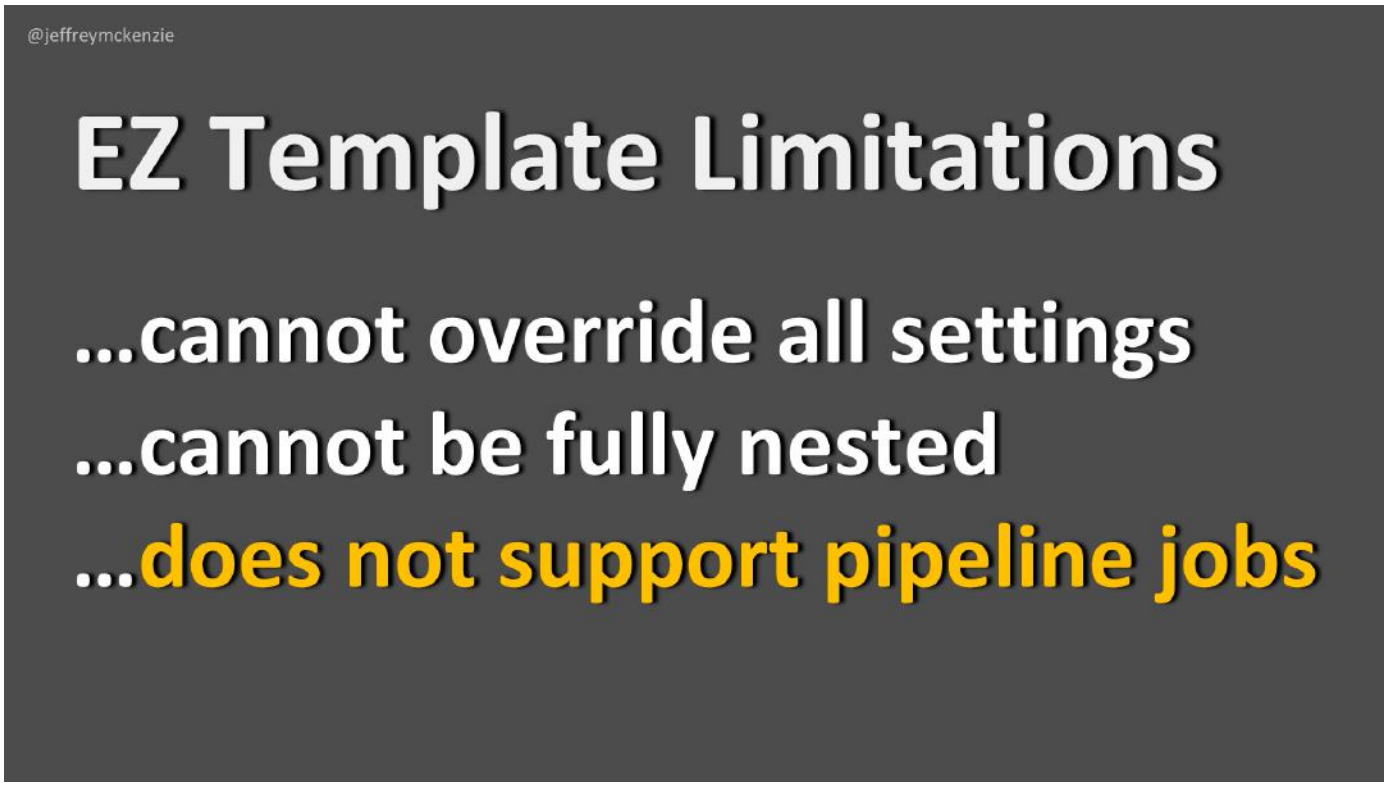


Next, you can't really nest the templates effectively.

Would be nice to have a base template,

And have another template inherit from that.

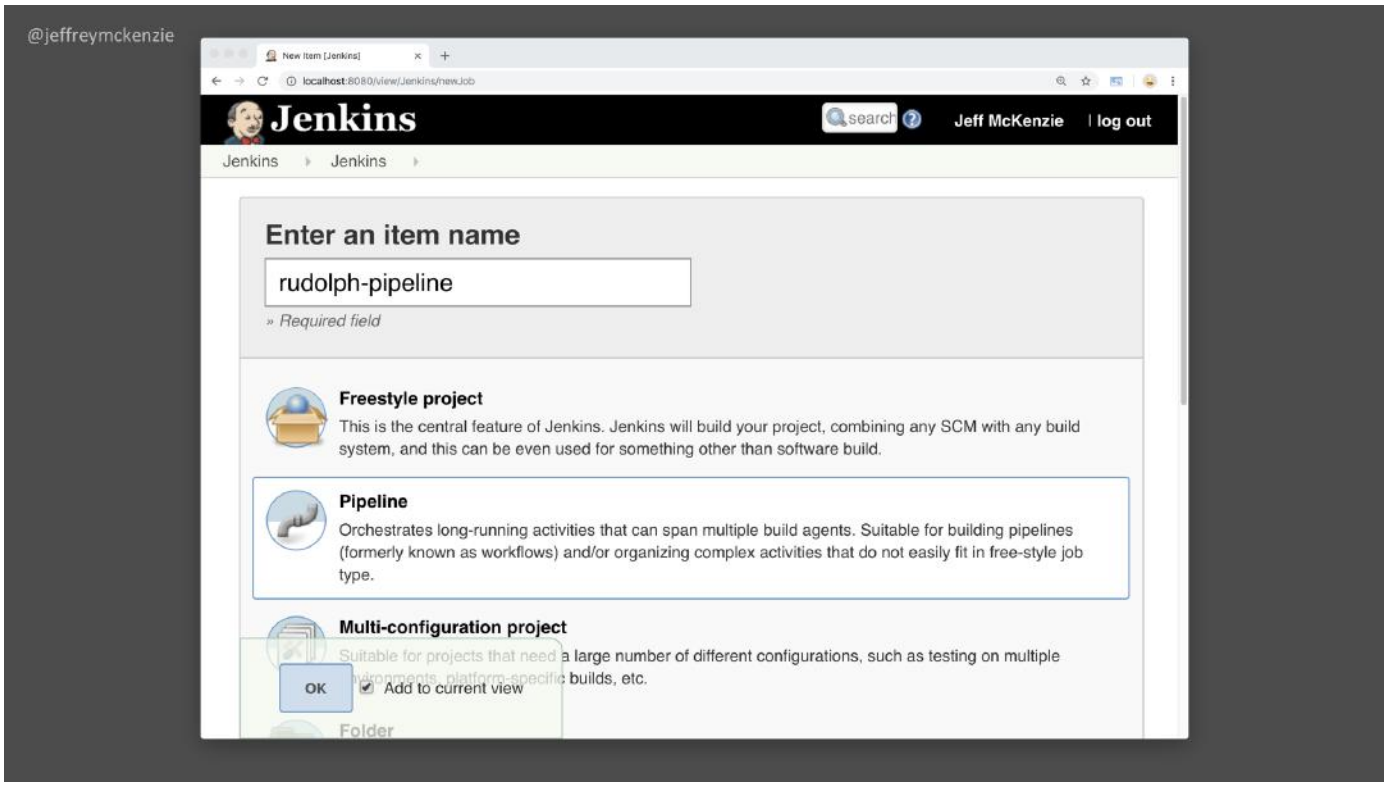
But you can't, for example, add parameters to the child template,
Which would make it more useful.



And lastly, templates only support freestyle jobs,
Not pipeline jobs.

Pipeline jobs is where the real power of Jenkins starts to come in.

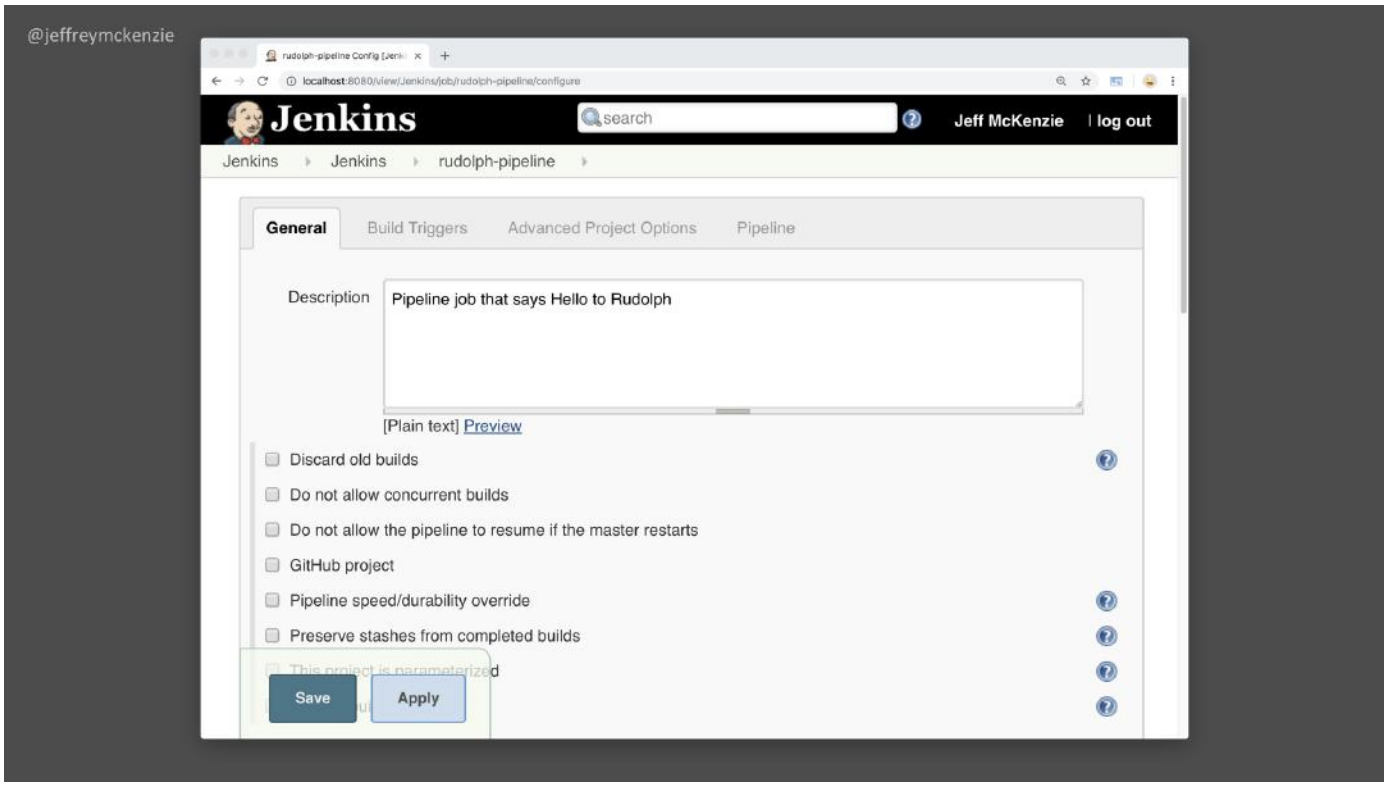
So templates are a great option where you have a lot of freestyle jobs
That exhibit a similar pattern, and where parameters would be helpful.



So let’s turn to our next option for scaling,
The pipeline job.

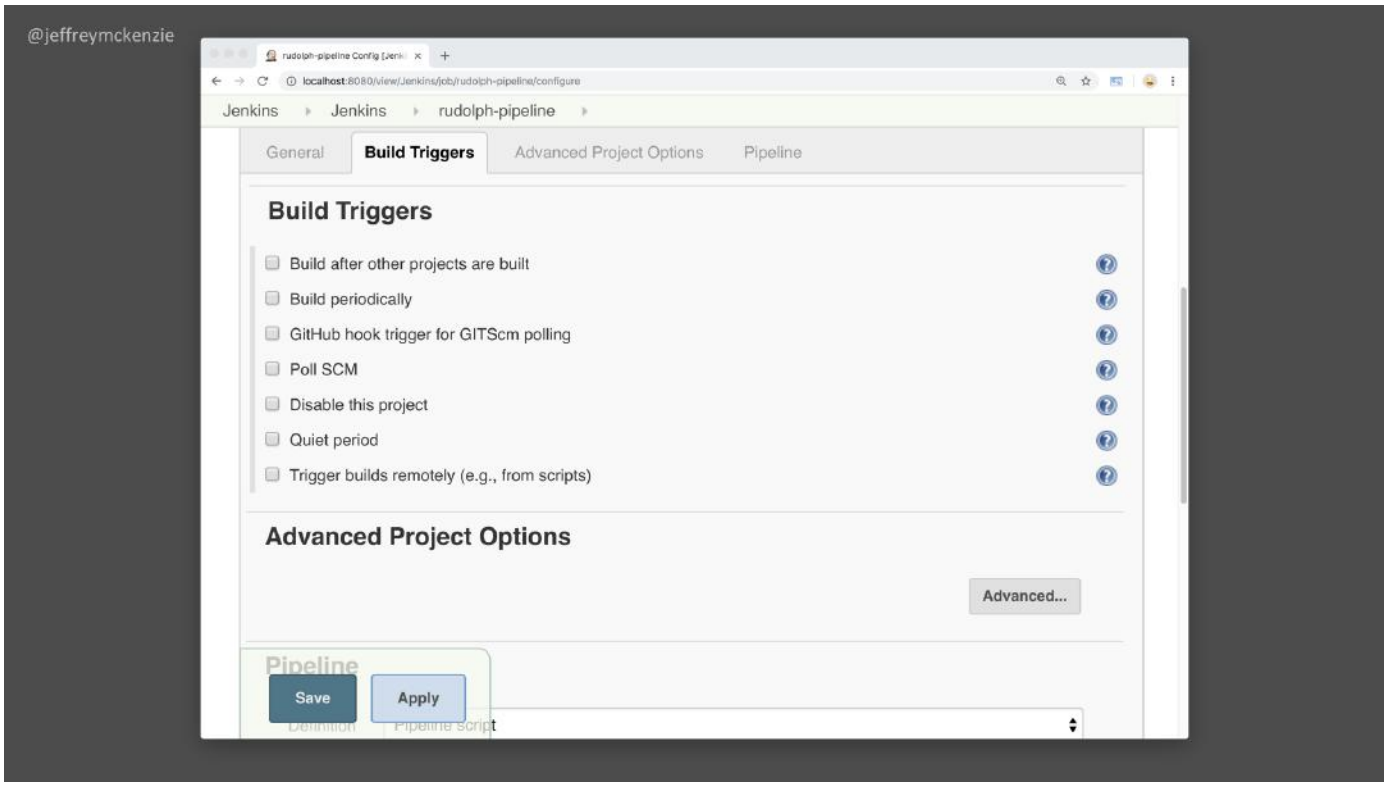
Let’s create one called Rudolph pipeline
that does the same thing
Our freestyle job does, to show the differences.

-- comment on pipeline description



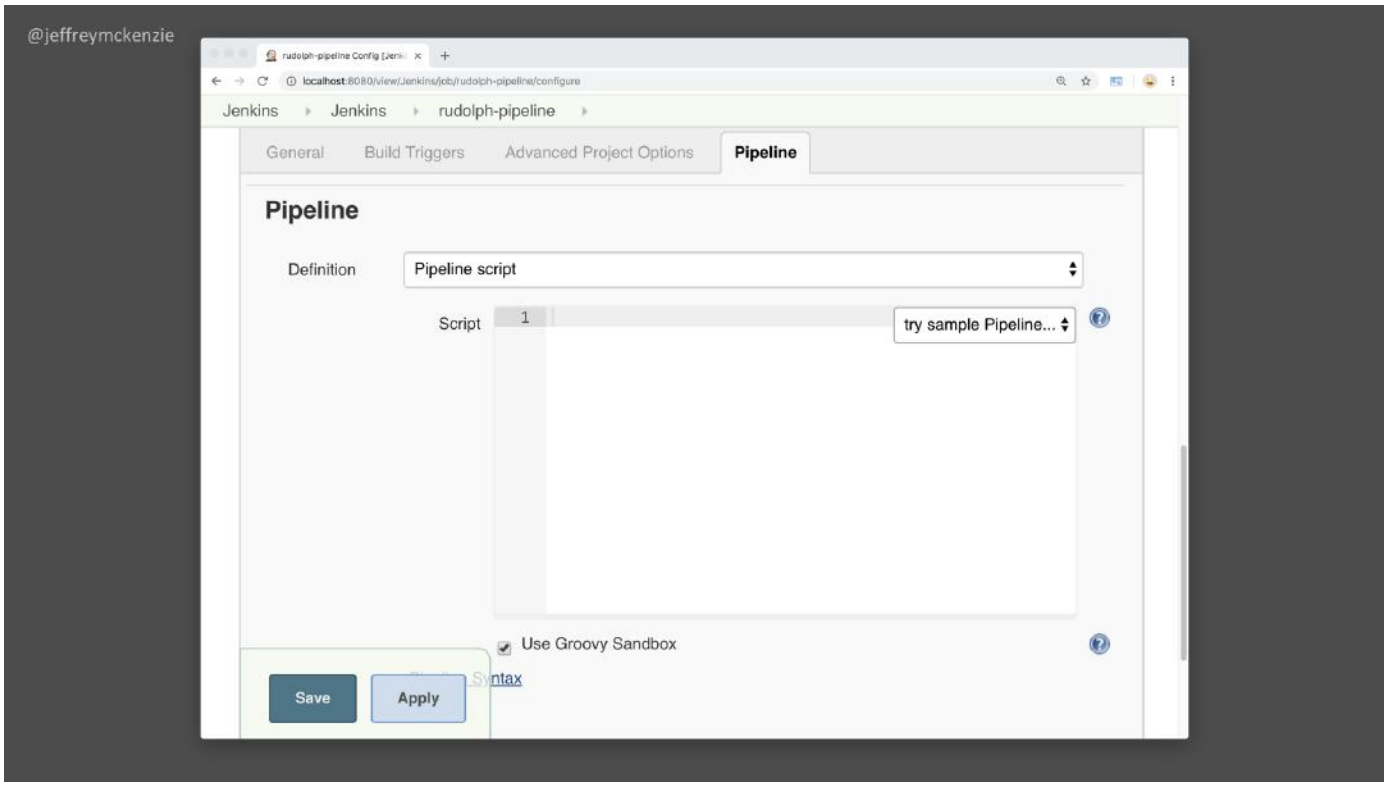
We will add our description,
And notice that this has less configuration options
Than a freestyle build.

We have the General section…



The build triggers section,
And the advanced project options.

But the main part of the pipeline job is the script



Let’s take a close-up look at that script

@jeffreymckenzie

```
pipeline {
  agent any
  options {
    buildDiscarder(logRotator(numToKeepStr:'3'))
  }
  parameters {
    string(
      name: 'GREETING',
      defaultValue: 'Hello',
      description: 'The way to say hello.'
    )
    string(
      name: 'REINDEER',
      defaultValue: 'Rudolph',
      description: 'The reindeer to say hello to.'
    )
  }
  stages {
    stage('Say Hi to Rudolph') {
      steps {
        checkout scm: [
          $class: 'GitSCM',
          branches: [
            [name: '*/master']
          ],
          userRemoteConfigs: [
            [url: 'file:///Users/jmckenzie/projects/santa']
          ]
        ]
        sh 'sh reindeer.sh "${GREETING}" "${REINDEER}"'
      }
    }
  }
}
```

Here’s the whole thing – As you can see, not a lot of code here

let’s break this down

@jeffreymckenzie

```
pipeline {
   agent any
   options {
      buildDiscarder(logRotator(numToKeepStr:'3'))
   }
}
```

On the Jenkins site, there's a reference area
for all of these commands and syntax,
So you can look all this stuff up

A pipeline script has to start with the pipeline block.
Then you specify the agent, or where this is going to run

For our purposes this can run anywhere…

We can set the build history to keep the last 3 builds,
just like we did in the freestyle job

@jeffreymckenzie

```
pipeline {
   agent any
   options {
      buildDiscarder(logRotator(numToKeepStr:'3'))
   }
   parameters {
      string( name: 'GREETING',
              defaultValue: 'Hello',
              description: 'The way to say hello. ')
   }
}
```

Then we can add the greeting parameter,
Again, same settings and values

@jeffreymckenzie

```
pipeline {
   agent any
   options { … }
   parameters {
      …
      string( name: 'REINDEER',
              defaultValue: 'Rudolph',
              description: 'The reindeer to say hello to.')
   }
}
```

Add our reindeer parameter…

@jeffreymckenzie

```
pipeline {
   agent any
   options { … }
   parameters { … }
   stages {
      stage('Say Hi to Rudolph') {
         steps {
         }
      }
   }
}
```

Then we add the stages block,
Which is just a way to group commands into sections.

We have only one stage, and we can give it a name…
And within a stage, we have steps.

@jeffreymckenzie

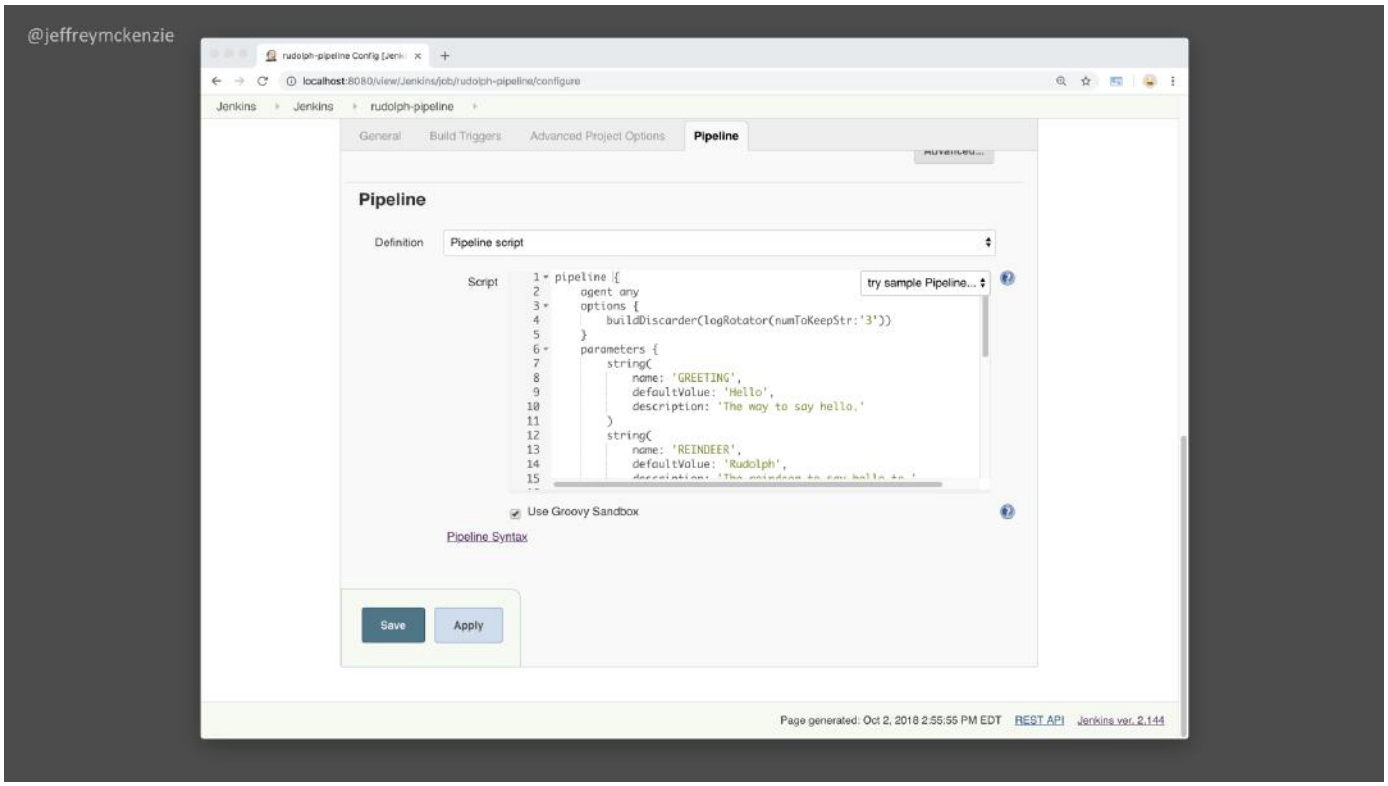
```
pipeline {
  ...
  stage('Say Hi to Rudolph') {
    steps {
      checkout scm: [$class: 'GitSCM',
        branches: [[name: '*/master']],
        userRemoteConfigs: [[url:
          'file:///Users/jmckenzie/projects/santa']]]
    }
  }
}
```

Our first step will be do to the checkout
From the git repository

@jeffreymckenzie

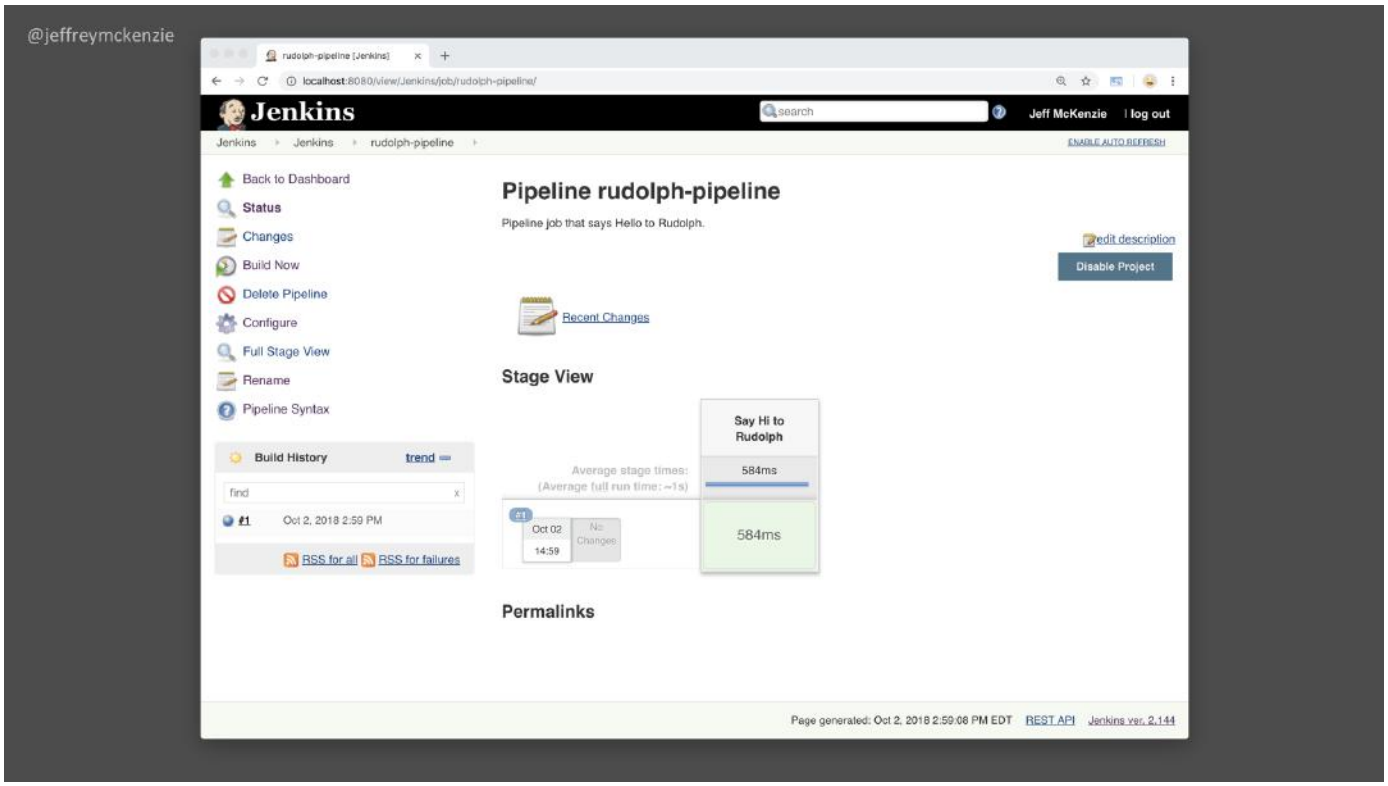
```
pipeline {
   ...
   stage('Say Hi to Rudolph') {
      steps {
         checkout scm:[ ... ]
         sh 'sh reindeer.sh
               "${GREETING}" "${REINDEER}"'
      }
   }
}
```

And the second step is to execute the shell script
Using the SH command and our parameters.

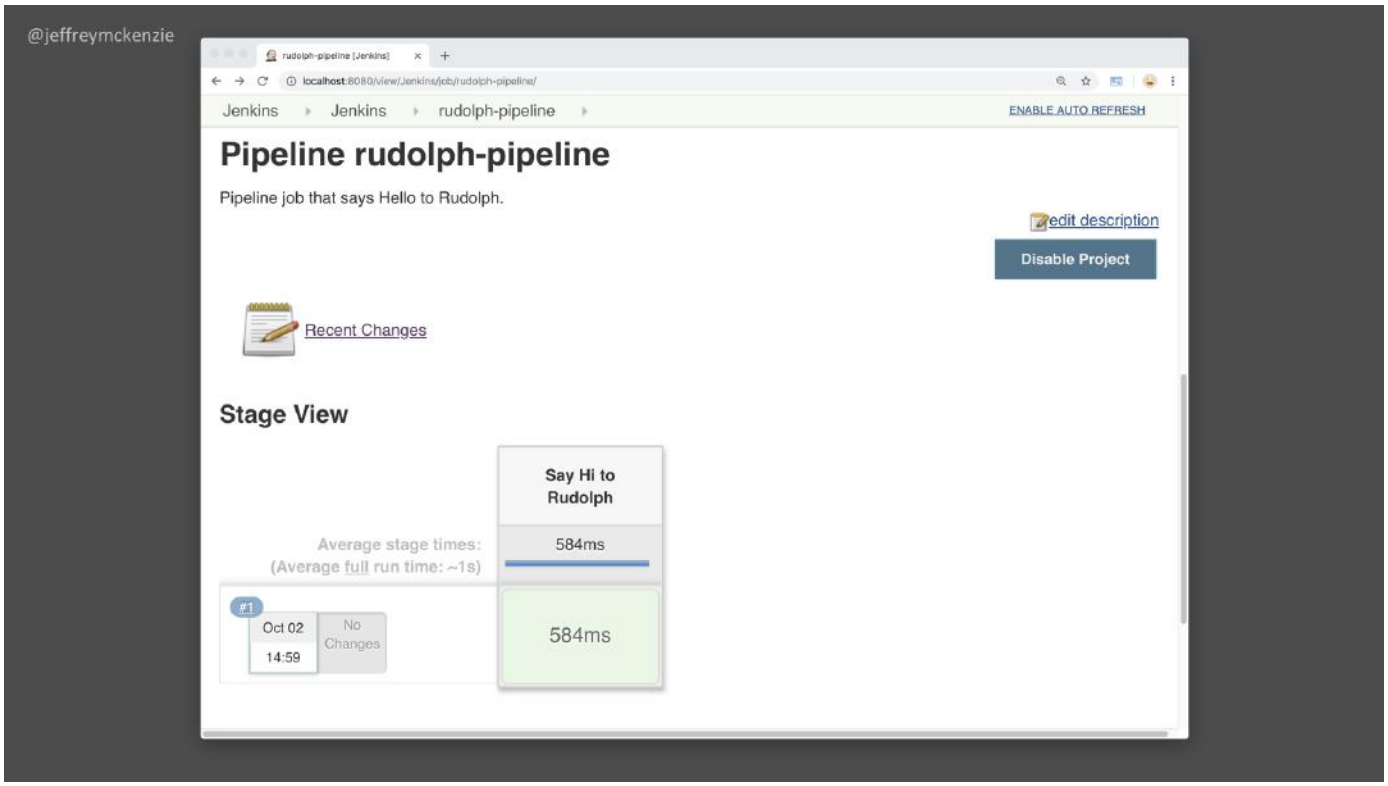


So here's what it looks like back in the pipeline project…

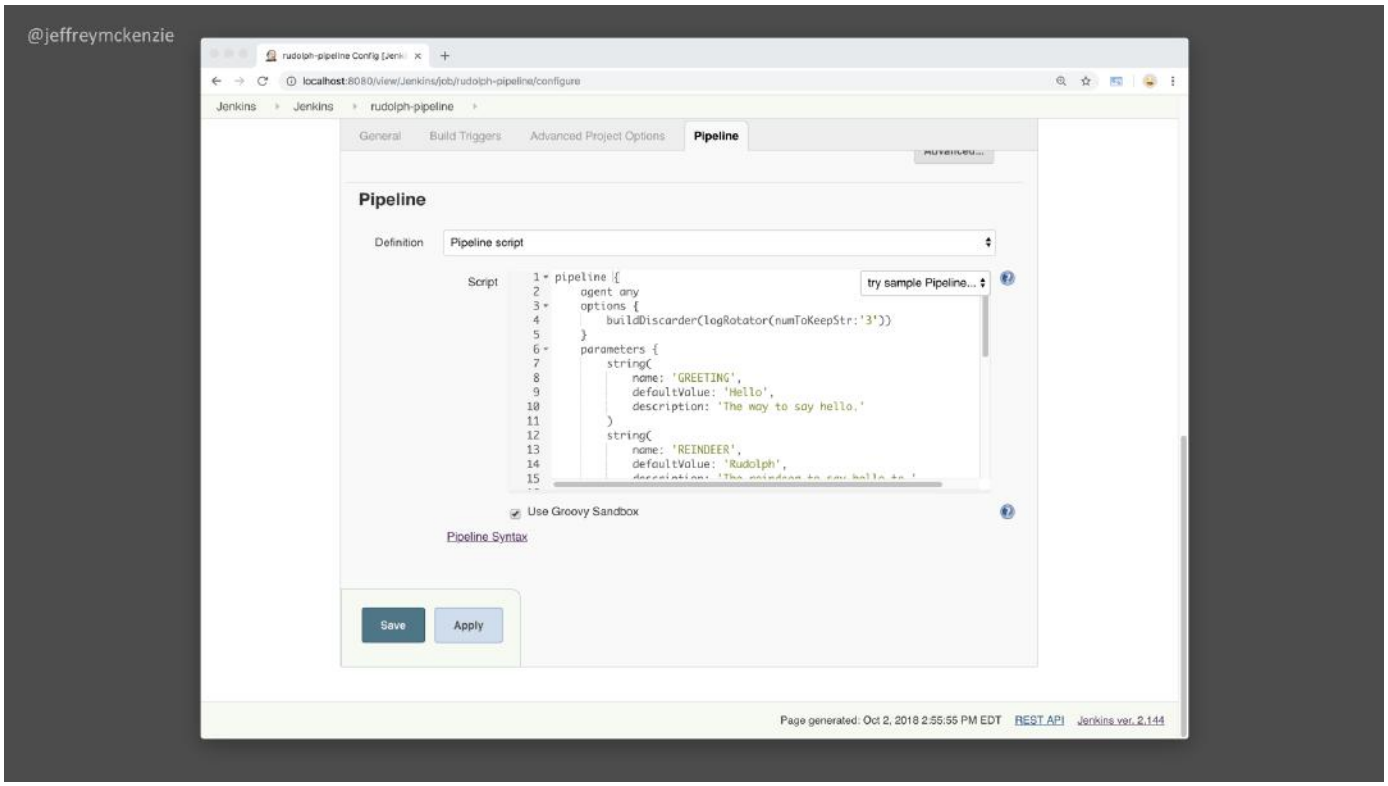
And let's run that and see what we get



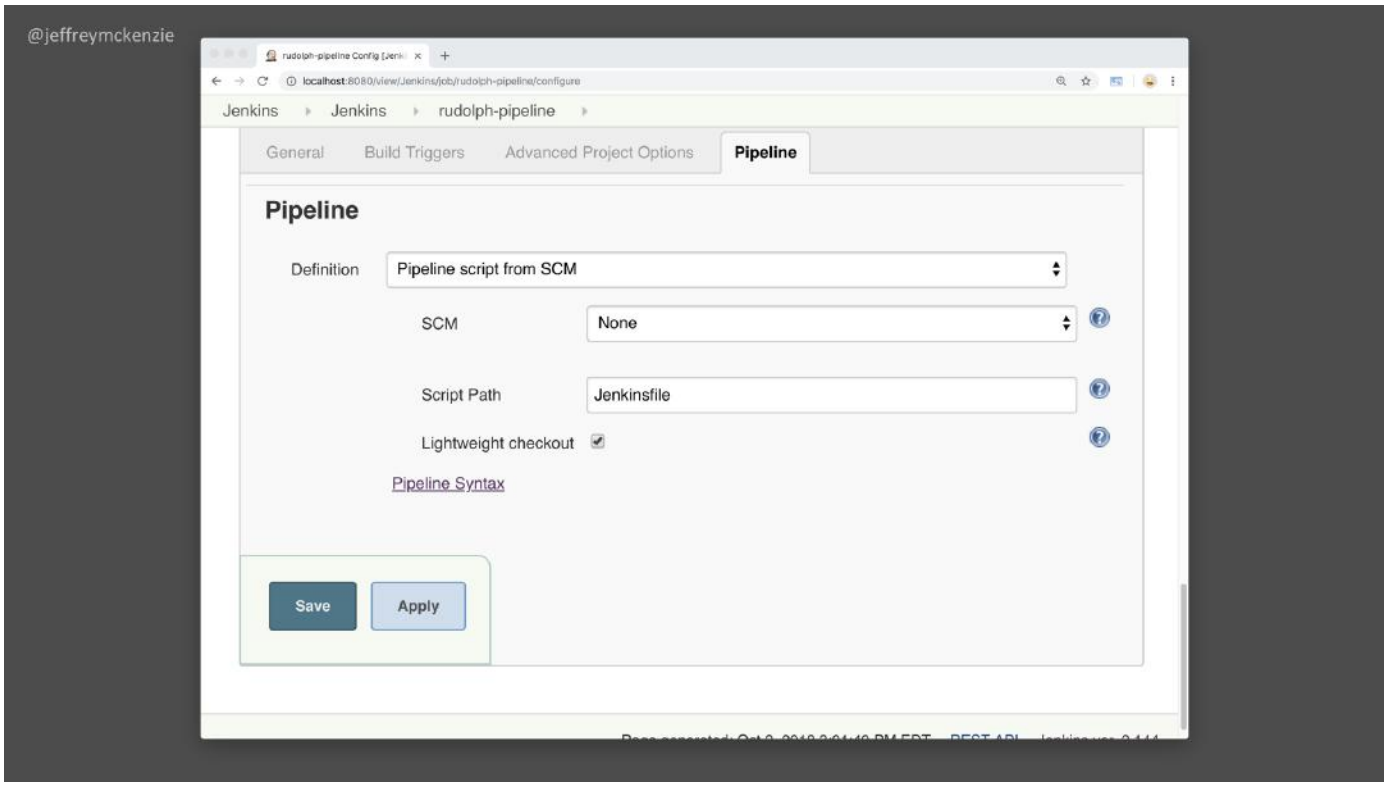
We have a successful build, and if you'll notice
There's a stage view here as well….



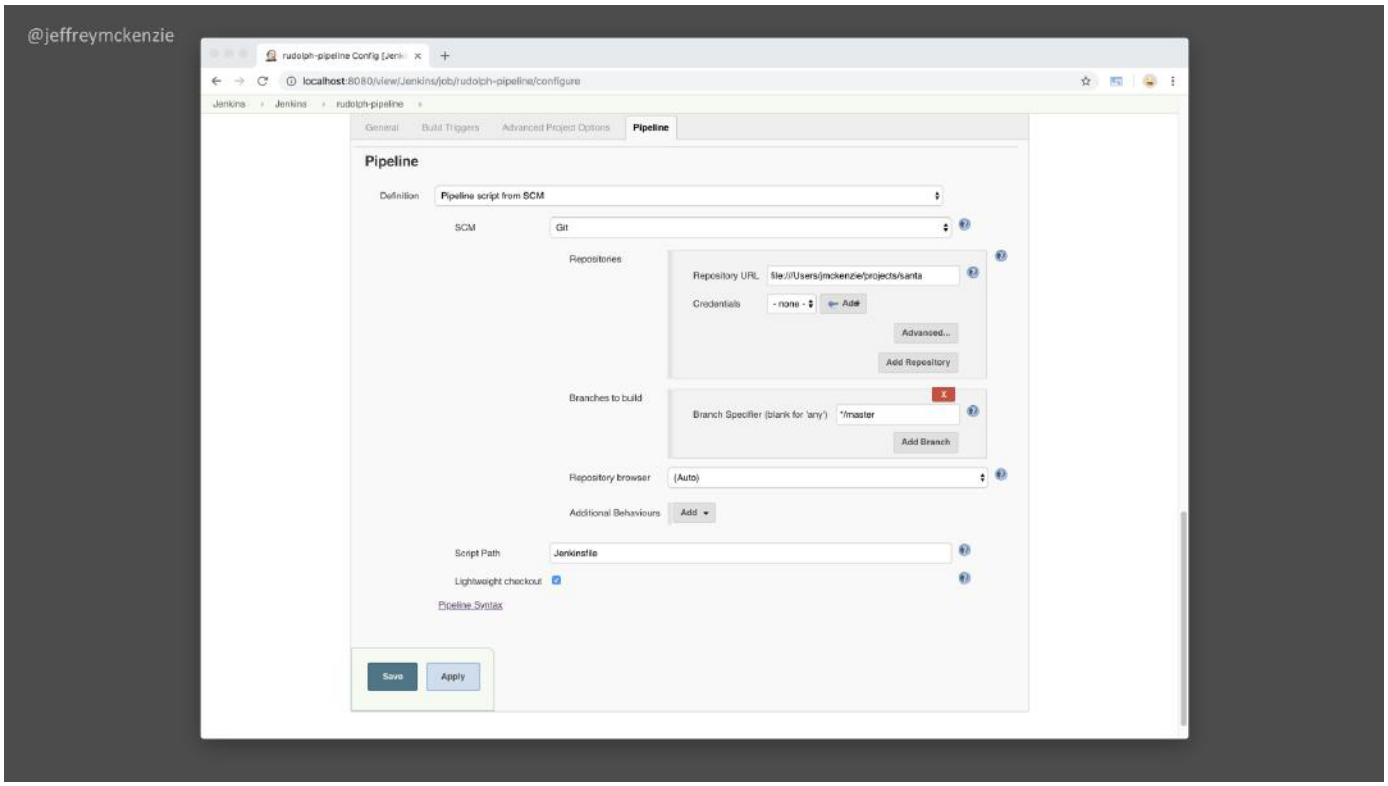
Which shows the name of the stage
And how long it took to run.



If we go back to our pipeline,
There’s another option here for definition…



Which is “pipeline script from SCM”
Or source control management,



We can choose a git repo where this is coming from,
Which is great because it allows us
To version control that pipeline script
Just like we would with any other code.

@jeffreymckenzie

```
pipeline {
  agent any
  options {
    buildDiscarder(logRotator(numToKeepStr:'3'))
  }
  parameters {
    string(
      name: 'GREETING',
      defaultValue: 'Hello',
      description: 'The way to say hello.'
    )
    string(
      name: 'REINDEER',
      defaultValue: 'Rudolph',
      description: 'The reindeer to say hello to.'
    )
  }
  stages {
    stage('Say Hi to Rudolph') {
      steps {
        checkout scm: [
          $class: 'GitSCM',
          branches: [
            [name: '*/master']
          ],
          userRemoteConfigs: [
            [url: 'file:///Users/jmckenzie/projects/santa']
          ]
        ]
        sh 'sh reindeer.sh "${GREETING}" "${REINDEER}"'
      }
    }
  }
}
```

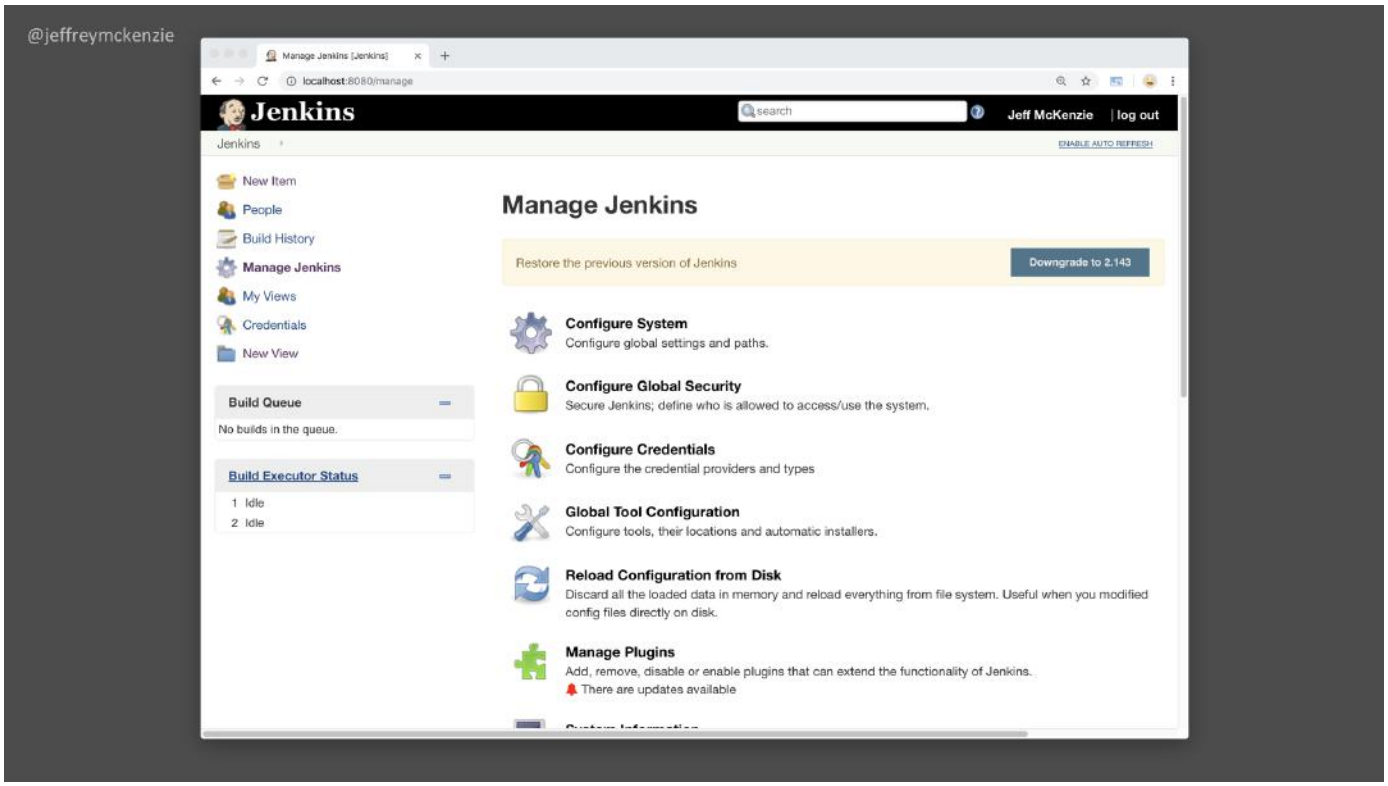
So that's great and everything,
But Santa has a problem –
wants to communicate with all his reindeer,

now only rudolph...

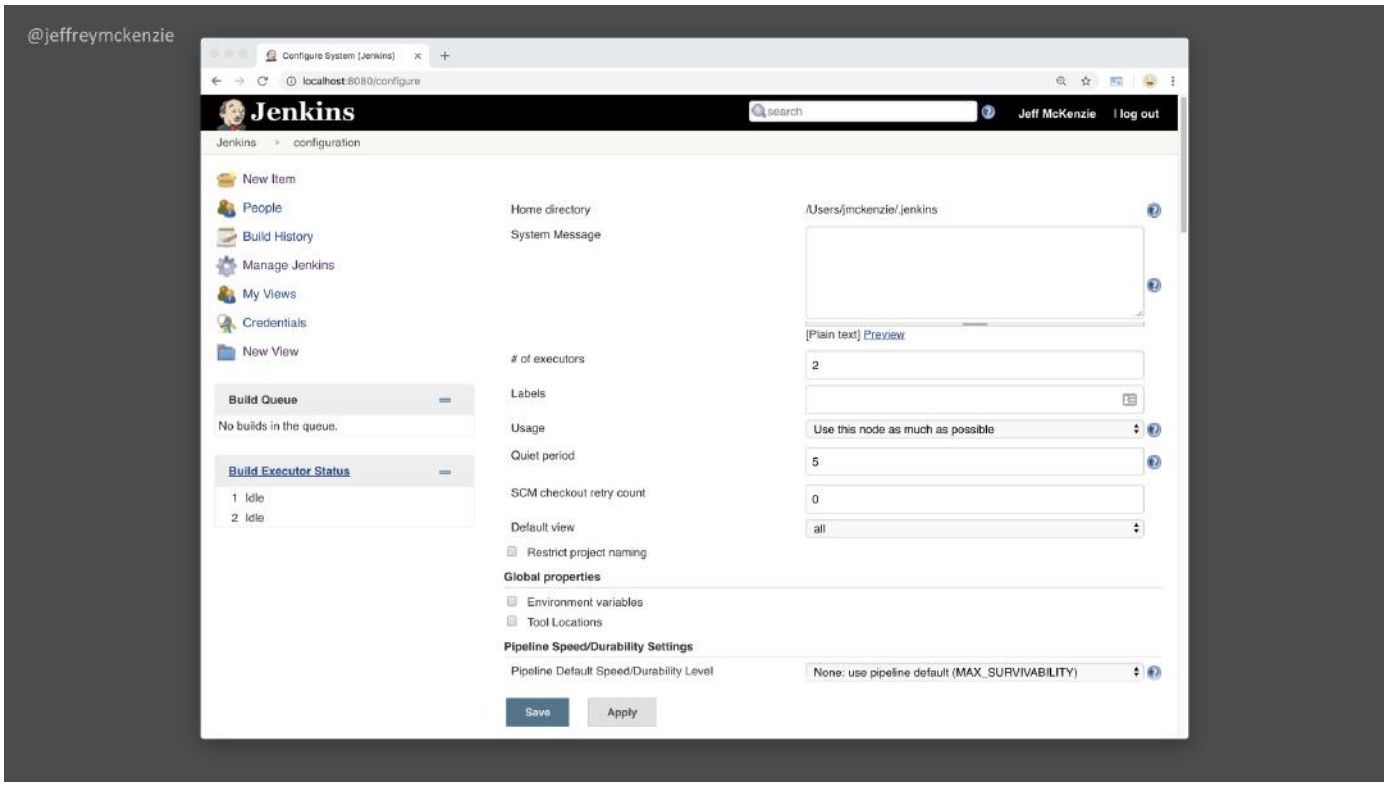
He has to copy this script
to any pipeline project that needs it.

So let’s fix that.

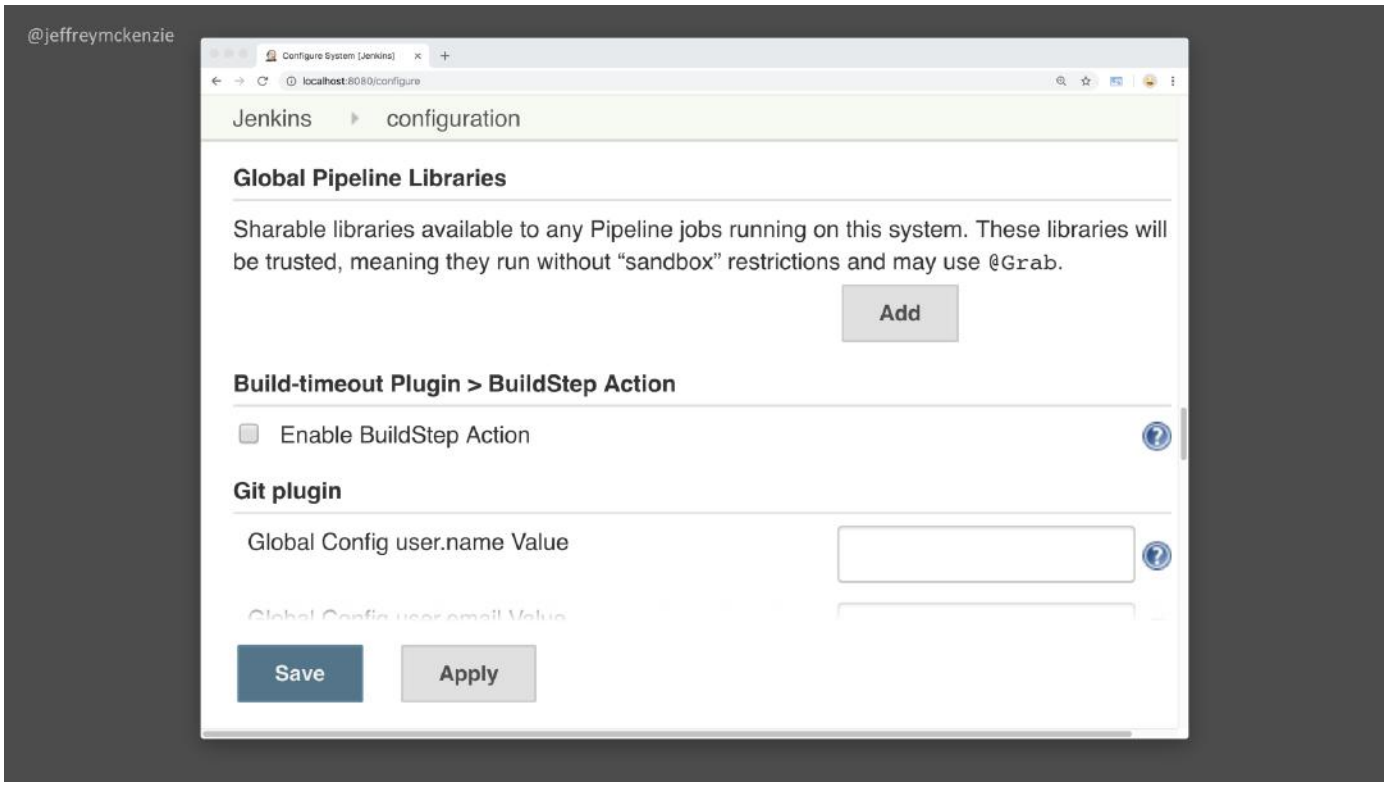
==========
https://commons.wikimedia.org/wiki/File:Mr_Santa_Claus_(HS85-10-30308).jpg
British Library [Public domain or Public domain], via Wikimedia Commons



If we go to the manage Jenkins page,
Into Configure System…

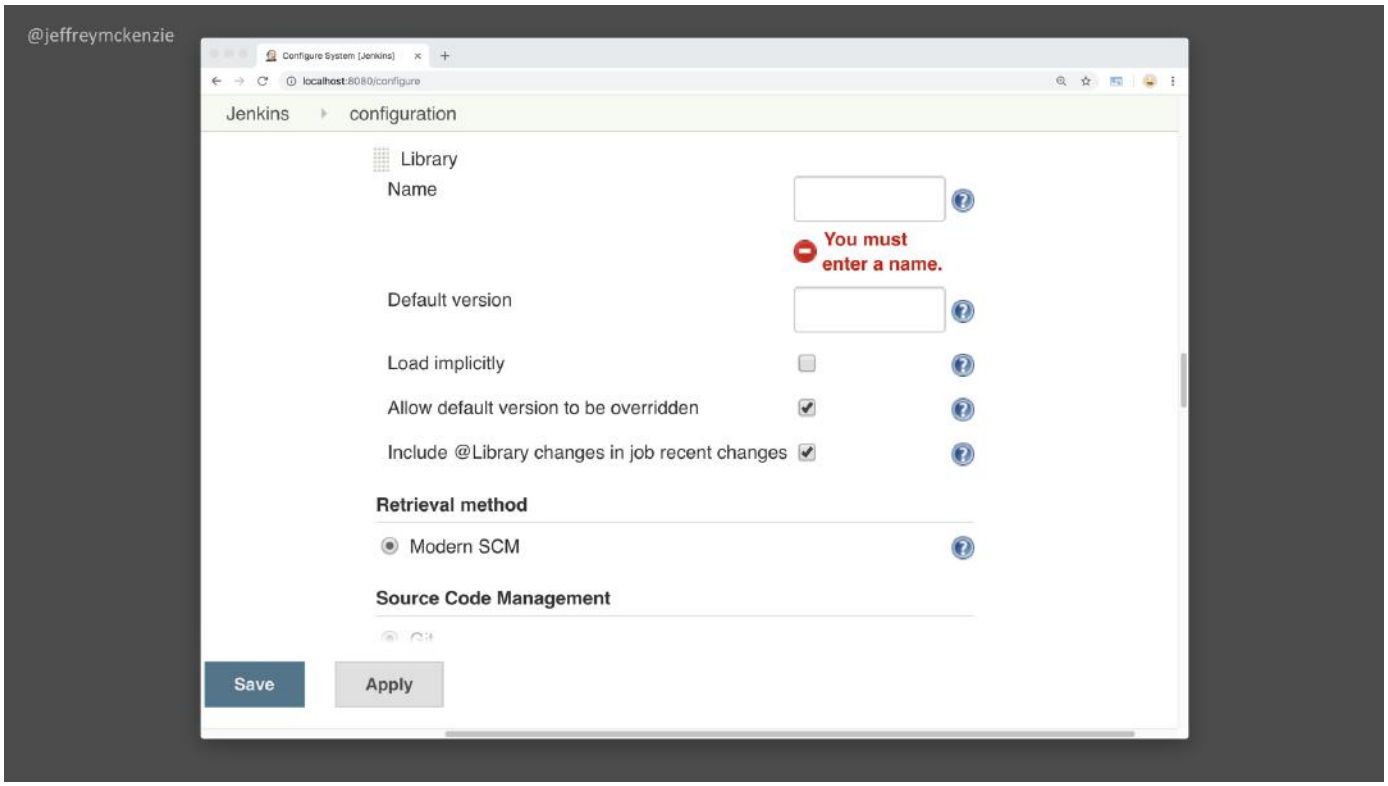


And we scroll down a bit,
We get to a section called Global Pipeline Libraries



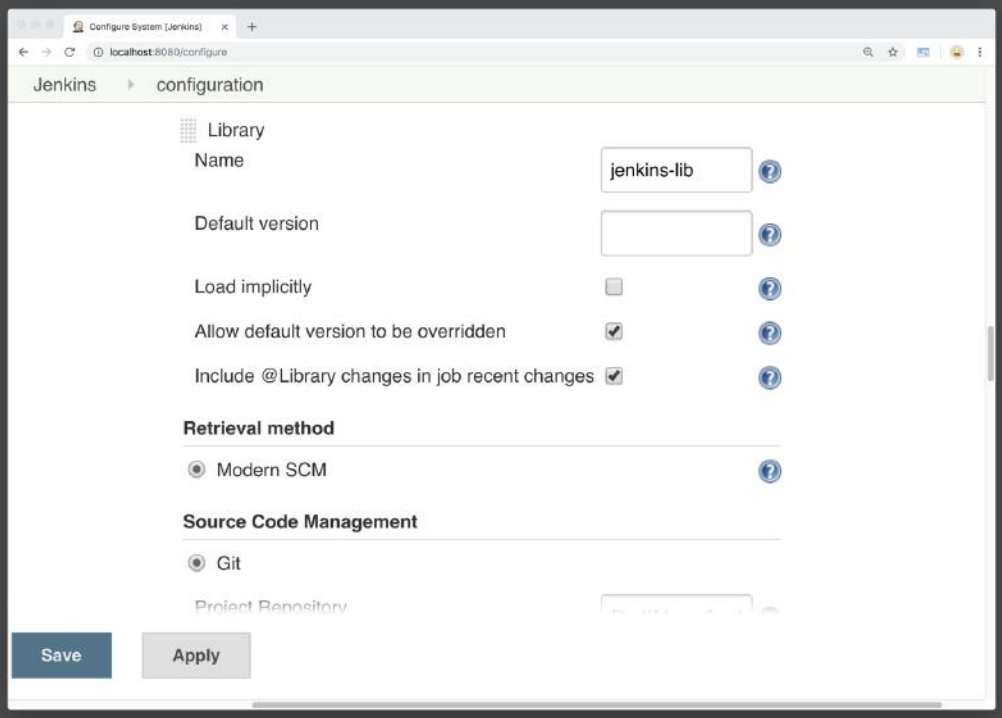
Which says…
Shareable libraries available to any Pipeline jobs
Running on this system.
That means we can create shared code
(written in groovy)
Accessible to any pipeline script.

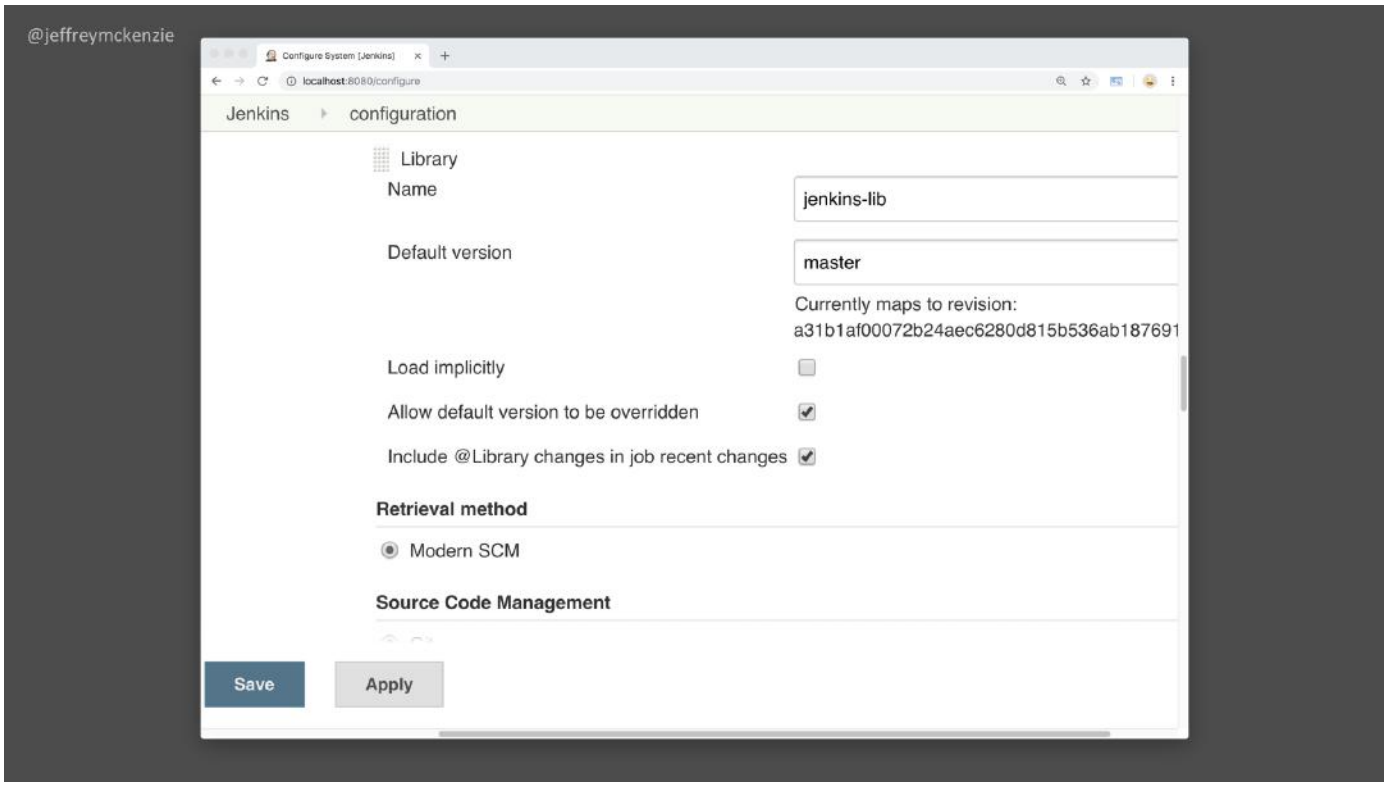
To do that, we click add…



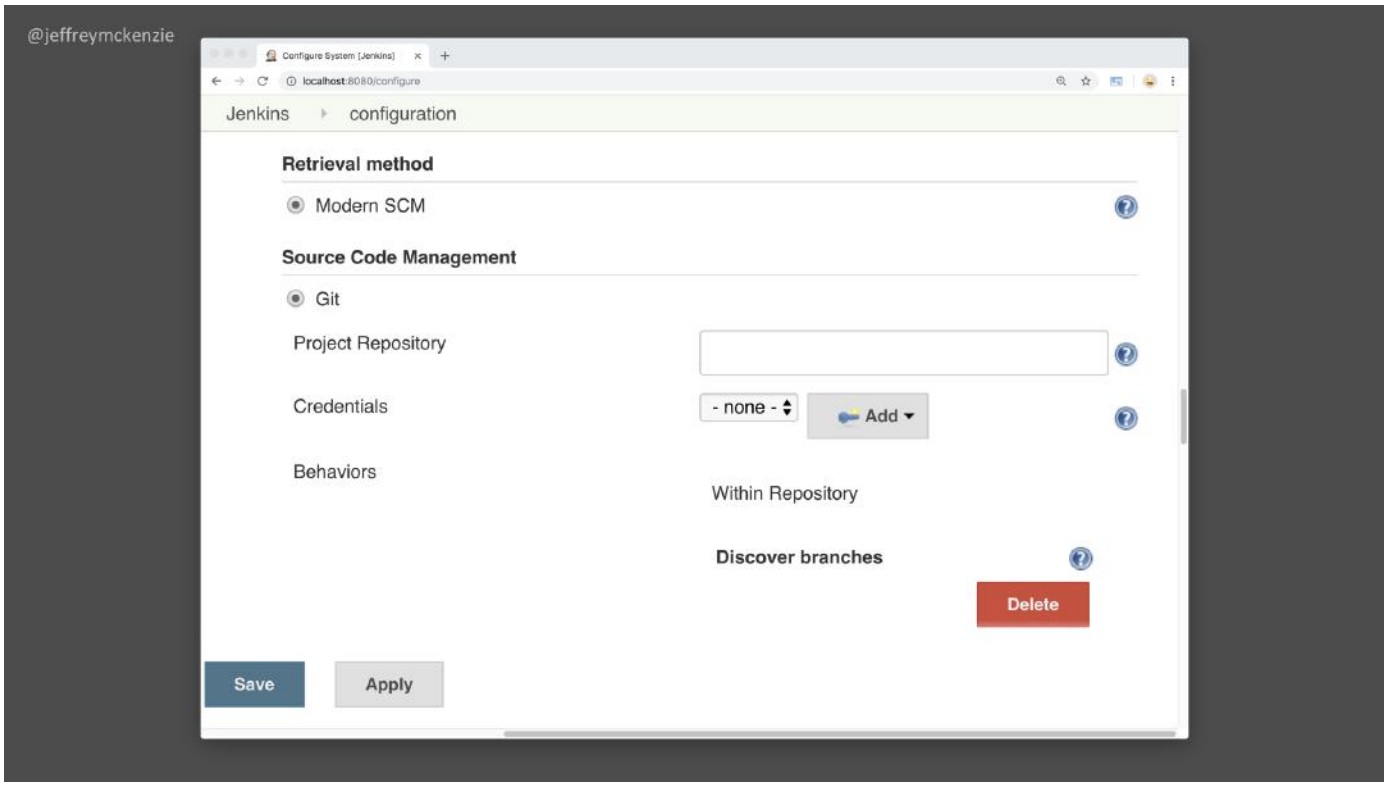
Then you'll add information about your library
-- a name: we'll call it Jenkins-lib

@jeffreymckenzie

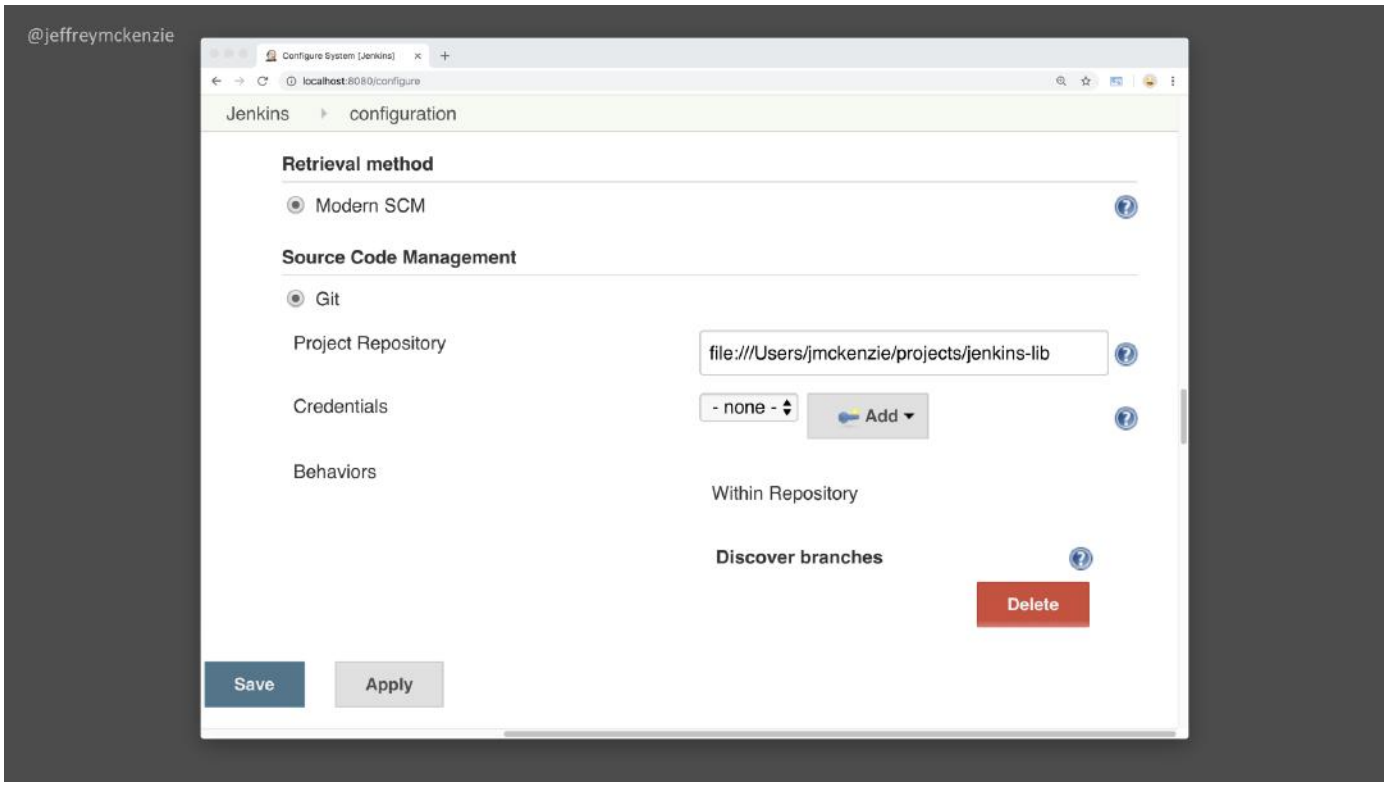




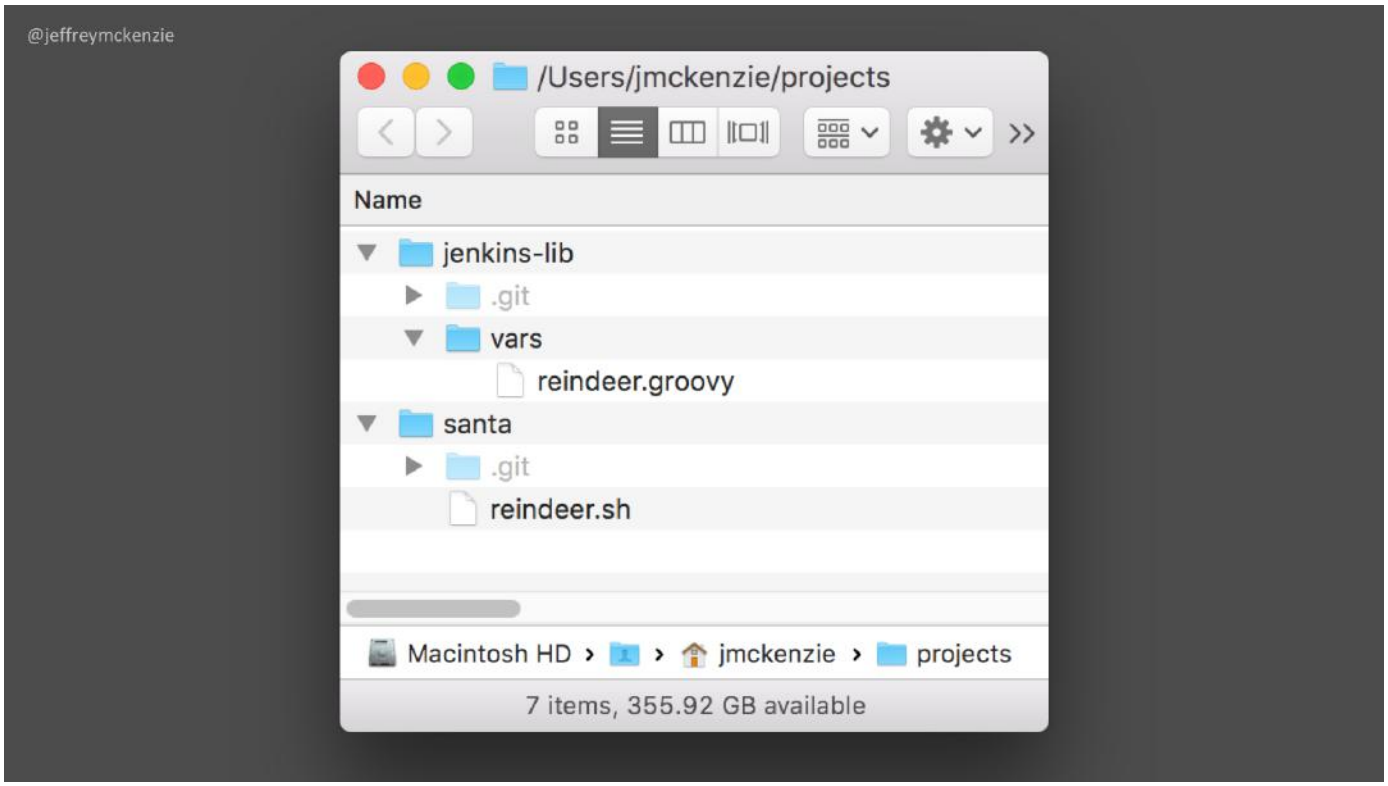
Default version refers to version control,
So we will just list the master branch



Then we add the git repo
we want to pull the shared code from



Again we'll point that to a separate git repo locally
And click save



So these are our 2 git repos –
Santa, which contains our shell script,

And Jenkins-lib, which contains
Our global pipeline library

Jenkins expects to see a directory
Called VARS, with groovy scripts
underneath that

-- let's look at the reindeer groovy script
in the library

@jeffreymckenzie

```
- reindeer.groovy

#!/usr/bin/env groovy
def call(Map<String, Object> options) {
  try {
    node {
      String greeting = options."greeting".toString()
      String reindeer = options."reindeer".toString()
      stage("Say Hi to ${reindeer}") {
        checkout scm: [
          $class: 'GitSCM',
          branches: [[name: '*/master']],
          userRemoteConfigs: [[url:
            'file:///Users/jmckenzie/projects/santa']]]
        sh "sh reindeer.sh ${greeting} ${reindeer}"
      }
    }
  }
  catch (Throwable err) {
    throw err
  }
}
```

so this is a rewrite of our pipeline script in groovy
Groovy is a scripting language
Similar to ruby,
that runs on the JVM

This is even less code than our pipeline script.
Let’s look at it in more detail.

@jeffreymckenzie

```
- reindeer.groovy

#!/usr/bin/env groovy
def call(Map<String, Object> options) {
  try {
    node {

    }
  }
  catch (Throwable err) {
    throw err
  }
}
```

You create a method using the DEF keyword,
And then name your method “call” --
You refer to the method in Jenkins by the file name
Rather than the actual method name,
Which we will see in a little bit.

We’re going to pass in a Map object called “options”
Which is simply a key-value pair collection.
We can use try and catch,
And the node block in where we place
What will run in the pipeline

@jeffreymckenzie

```
- reindeer.groovy

#!/usr/bin/env groovy
def call(Map<String, Object> options) {
  try {
    node {
      String greeting =
          options."greeting".toString()
      String reindeer =
          options."reindeer".toString()
    }
  }
  catch (Throwable err) { ... }
}
```

Then we declare two variables –
Greeting and reindeer, that hold our parameters.

@jeffreymckenzie

```
- reindeer.groovy

def call(Map<String, Object> options) {
  ...
      String greeting = ...
      String reindeer = ...
      stage("Say Hi to ${reindeer}") {

      }
    }
  }
  catch (Throwable err) { ... }
}
```

Then we add the stage block –
And here we can actually incorporate
The variable name into the Jenkins output

@jeffreymckenzie

```
- reindeer.groovy

def call(Map<String, Object> options) {
  ...
  stage("Say Hi to ${reindeer}") {
    checkout scm: [
      $class: 'GitSCM',
      branches: [[name: '*/master']],
      userRemoteConfigs: [[url:'file:
///Users/jmckenzie/projects/santa']]]
  }
  ...
}
```

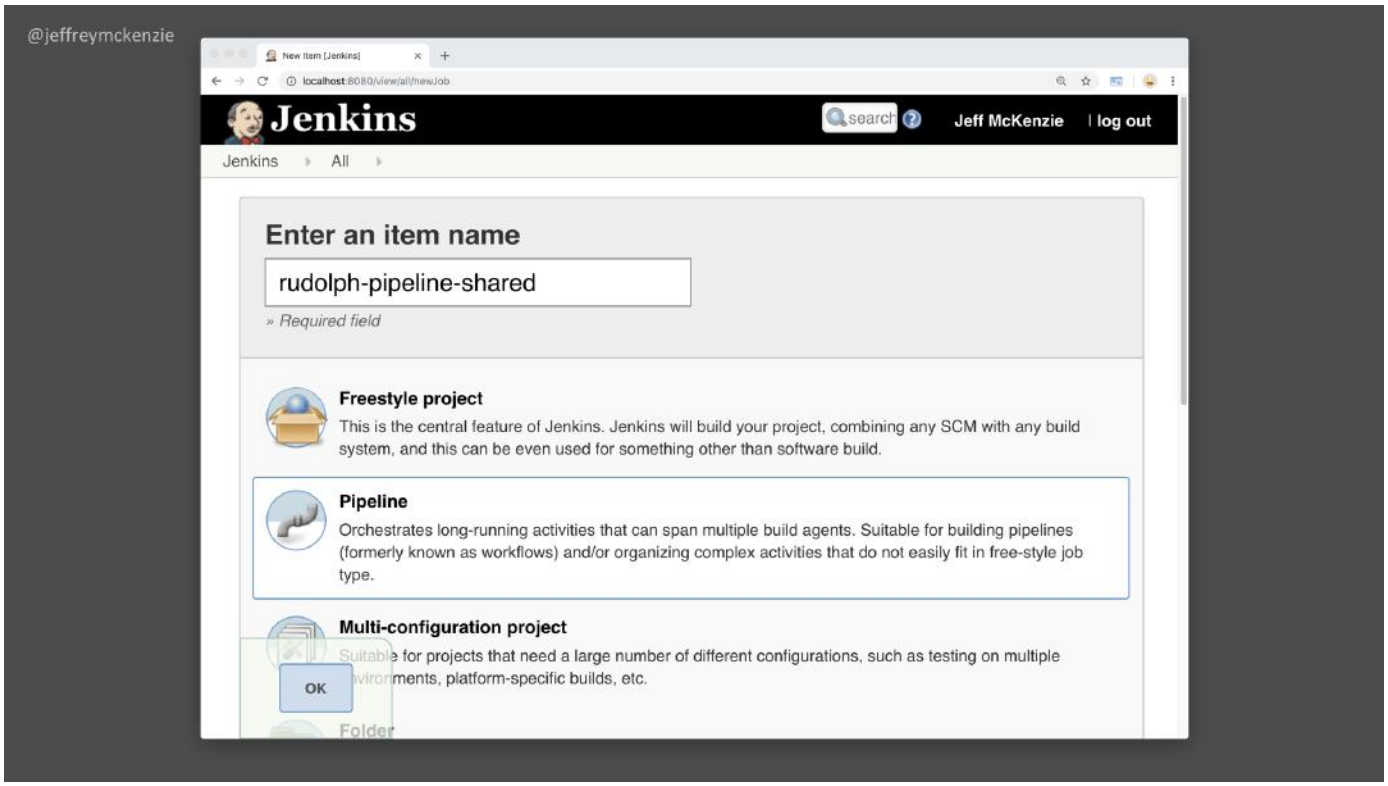
Then the git checkout from our santa repo,
to pull our reindeer shell script

@jeffreymckenzie

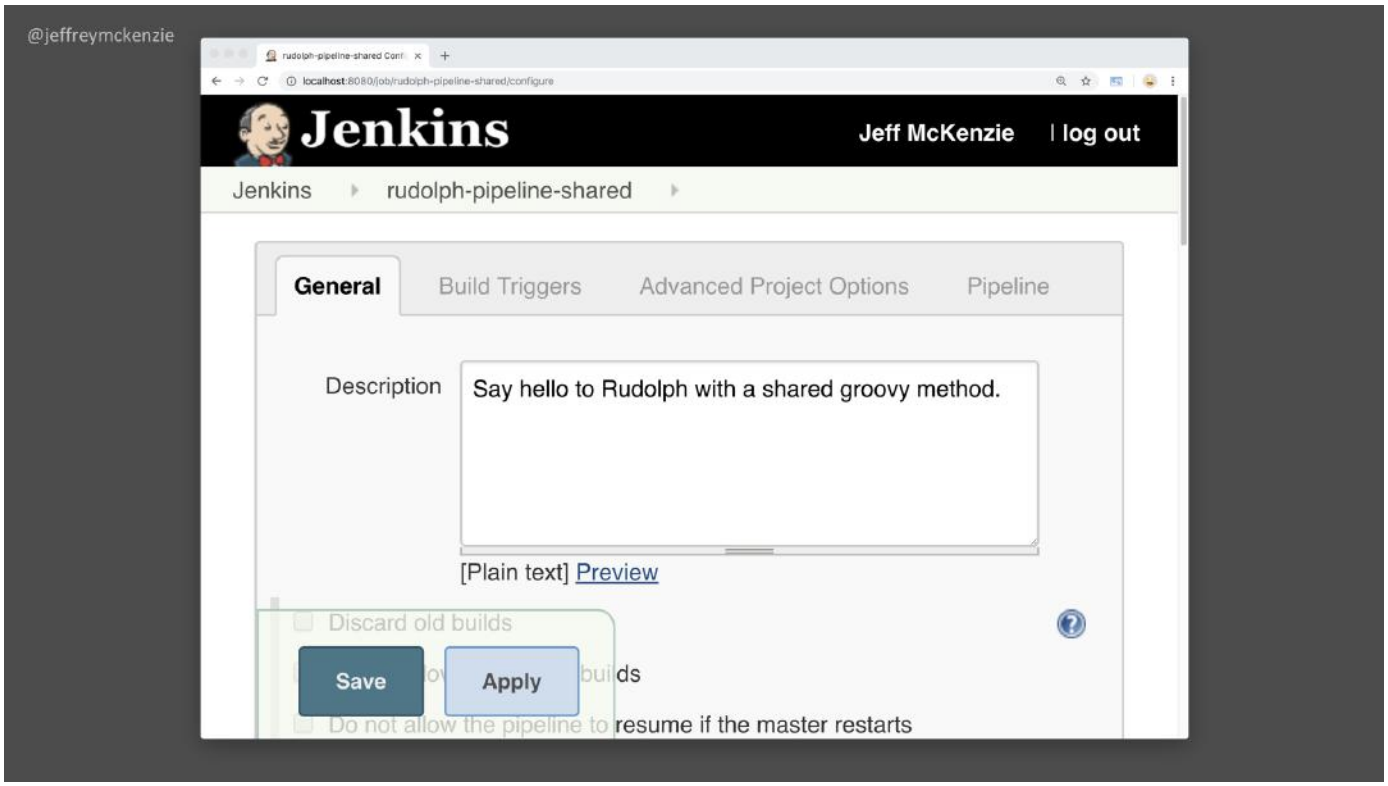
```
- reindeer.groovy

def call(Map<String, Object> options) {
  ...
  stage("Say Hi to ${reindeer}") {
    checkout scm: [ ... ]
    sh "sh reindeer.sh ${greeting}
                       ${reindeer}"

  }
  ...
}
```

And finally our call to the shell script,
Where we can pass the variables in.

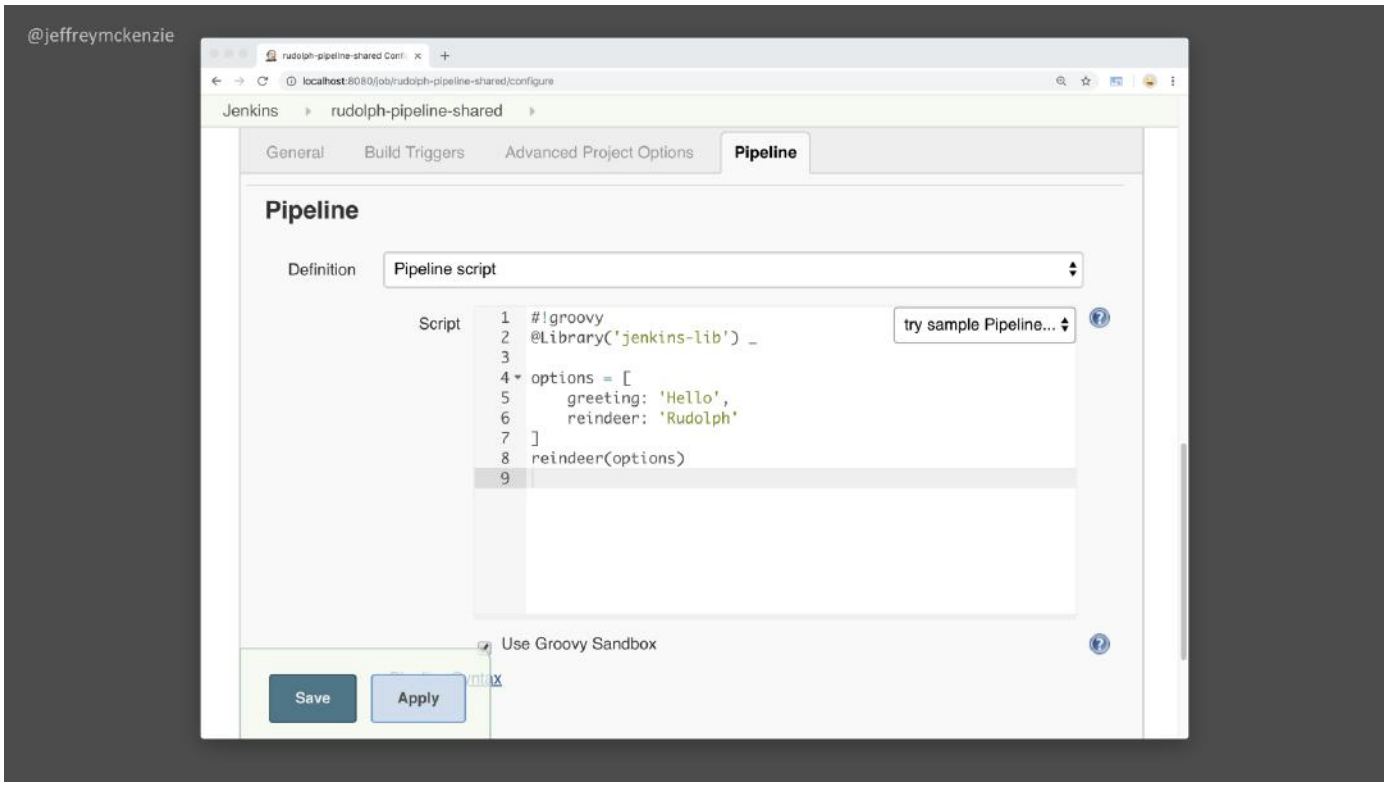
Let’s create a new pipeline job and see how this works.



We'll call it Rudolph-pipeline-shared



We're going to say hello to Rudolph with a shared groovy method.



We go down to the pipeline script area
And just add a few lines here –

And that's it –

Lets take a closer look.

@jeffreymckenzie

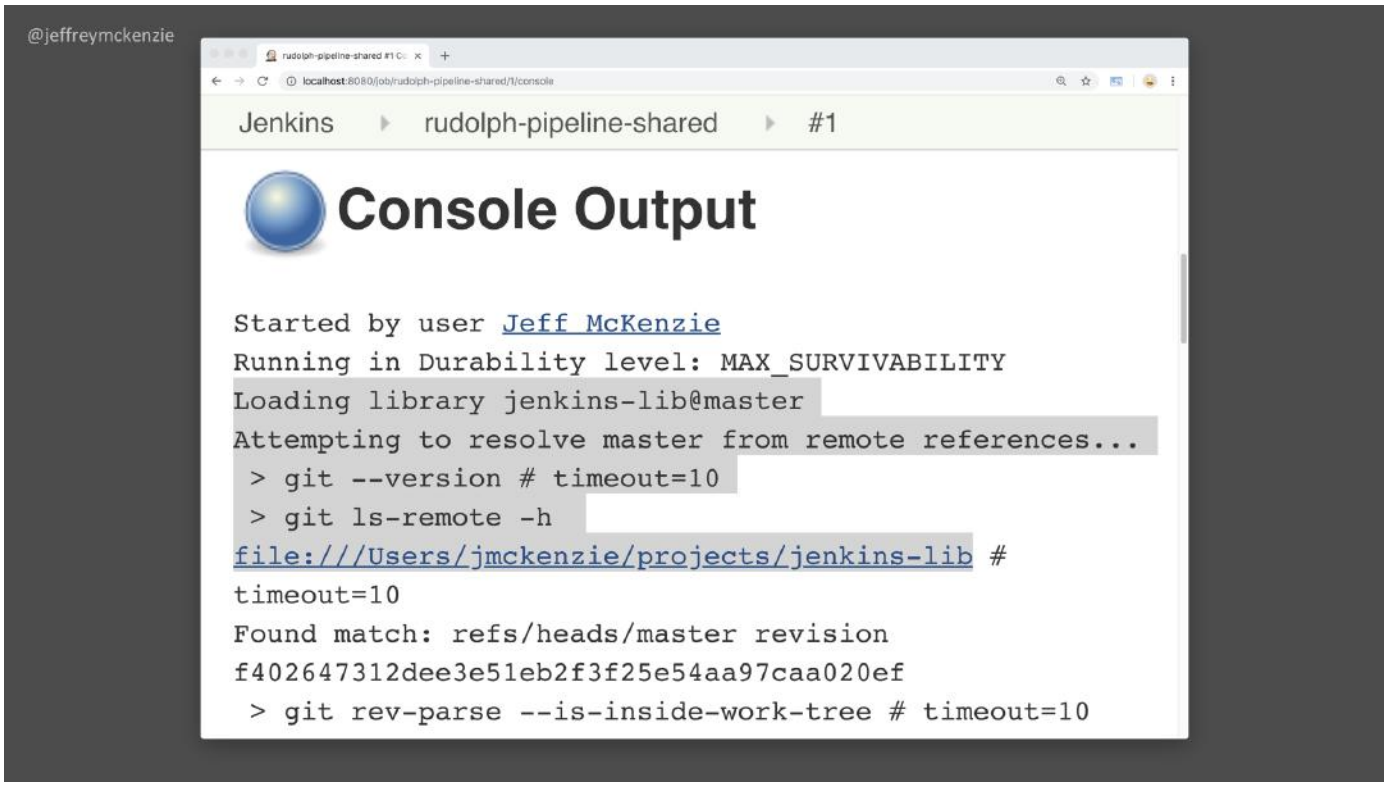
```
#!groovy
@Library('jenkins-lib') _

options = [
    greeting: 'Hello',
    reindeer: 'Rudolph'
]
reindeer(options)
```

The first line tells Jenkins you're executing groovy –
Second line is a reference to the global library
You want to use – (note underscore)

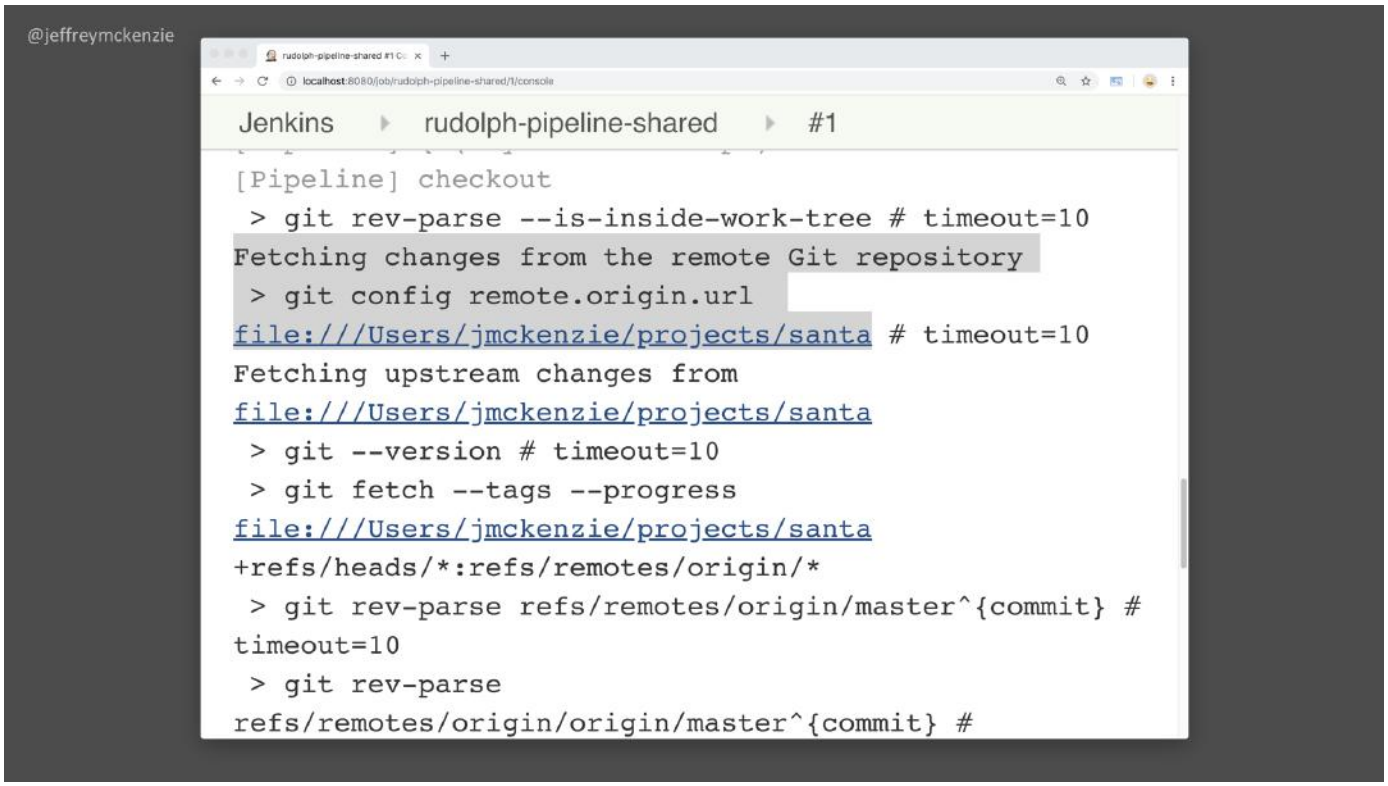
Then you create the map object with the variable values,

And call the method by passing in the options.
Jenkins will look for the file called reindeer
and execute the method in it

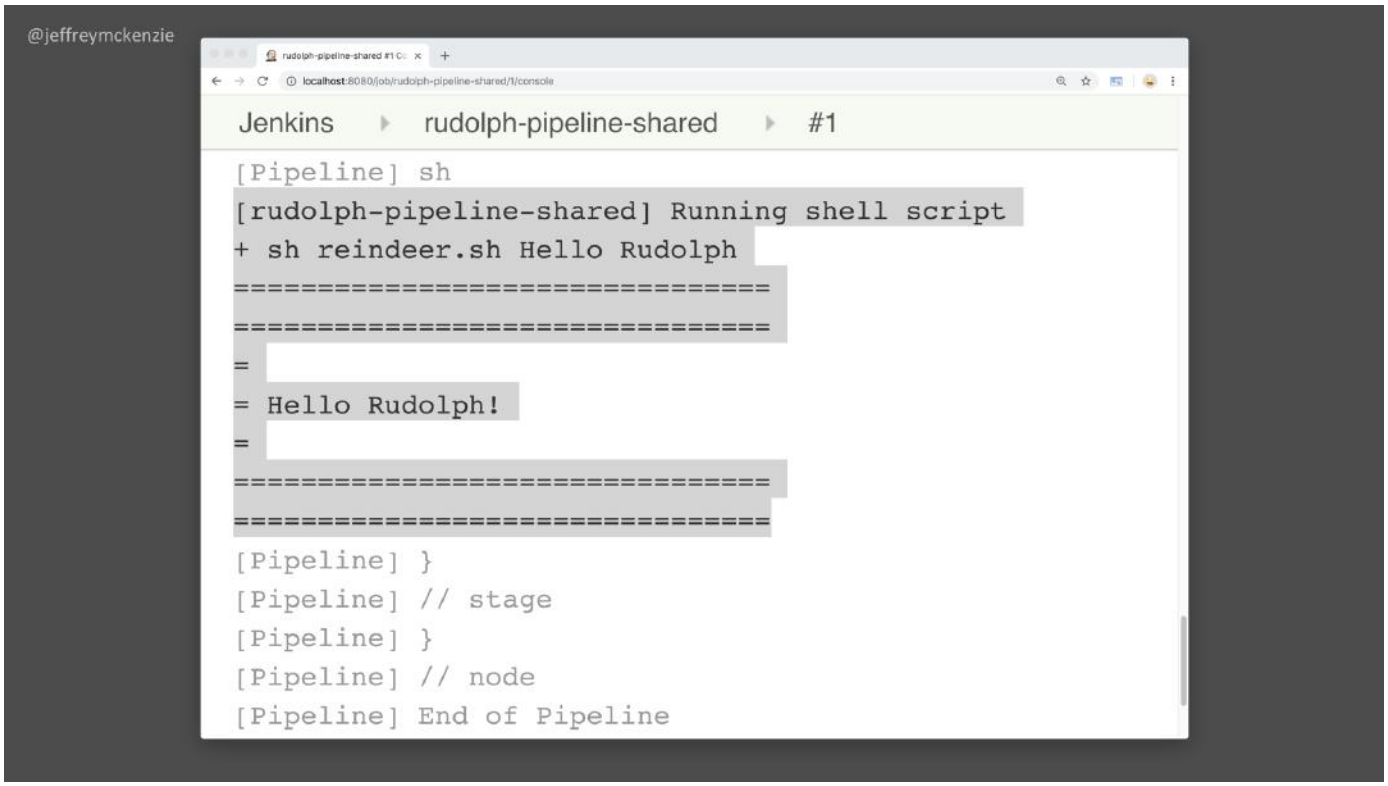
Let's run this and look at the result…



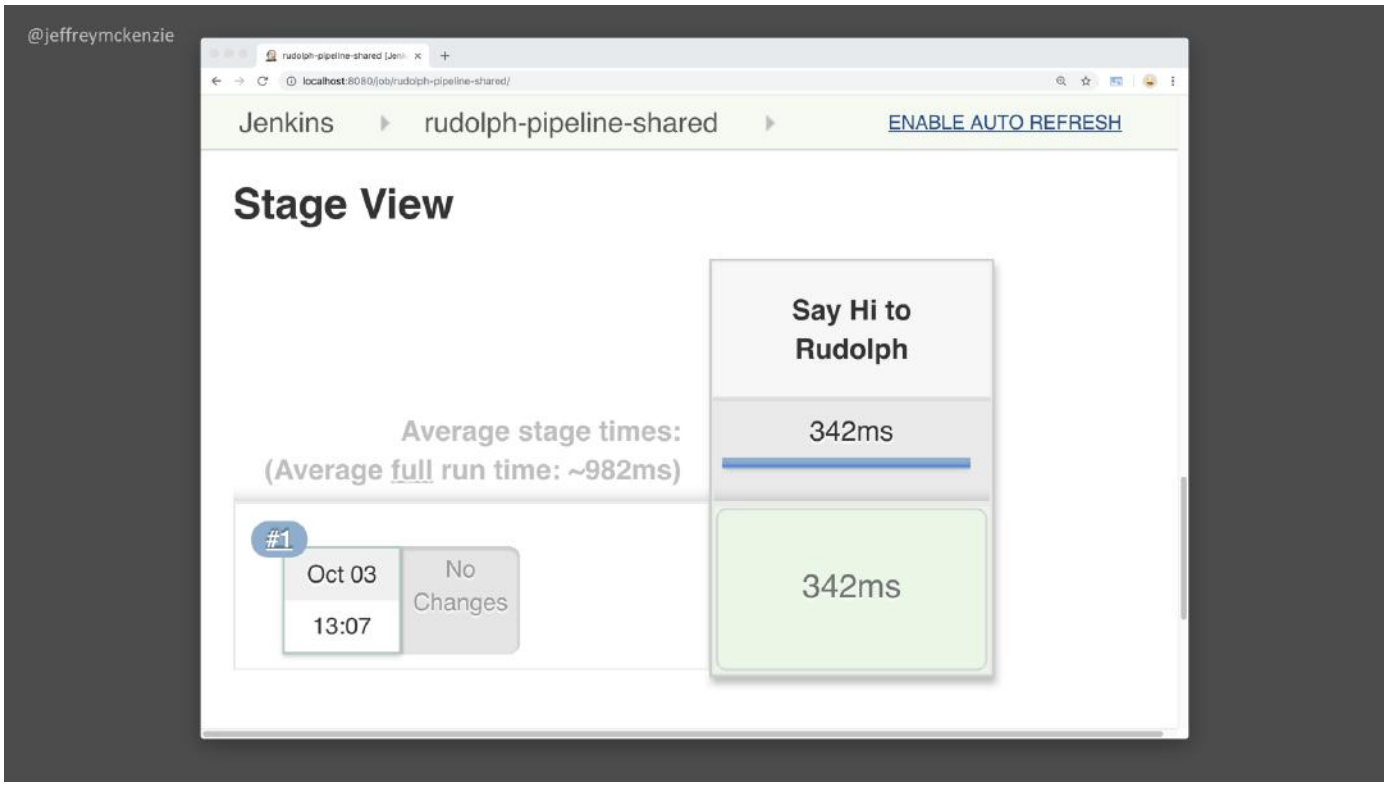
First you can see Jenkins is
loading the library from git…



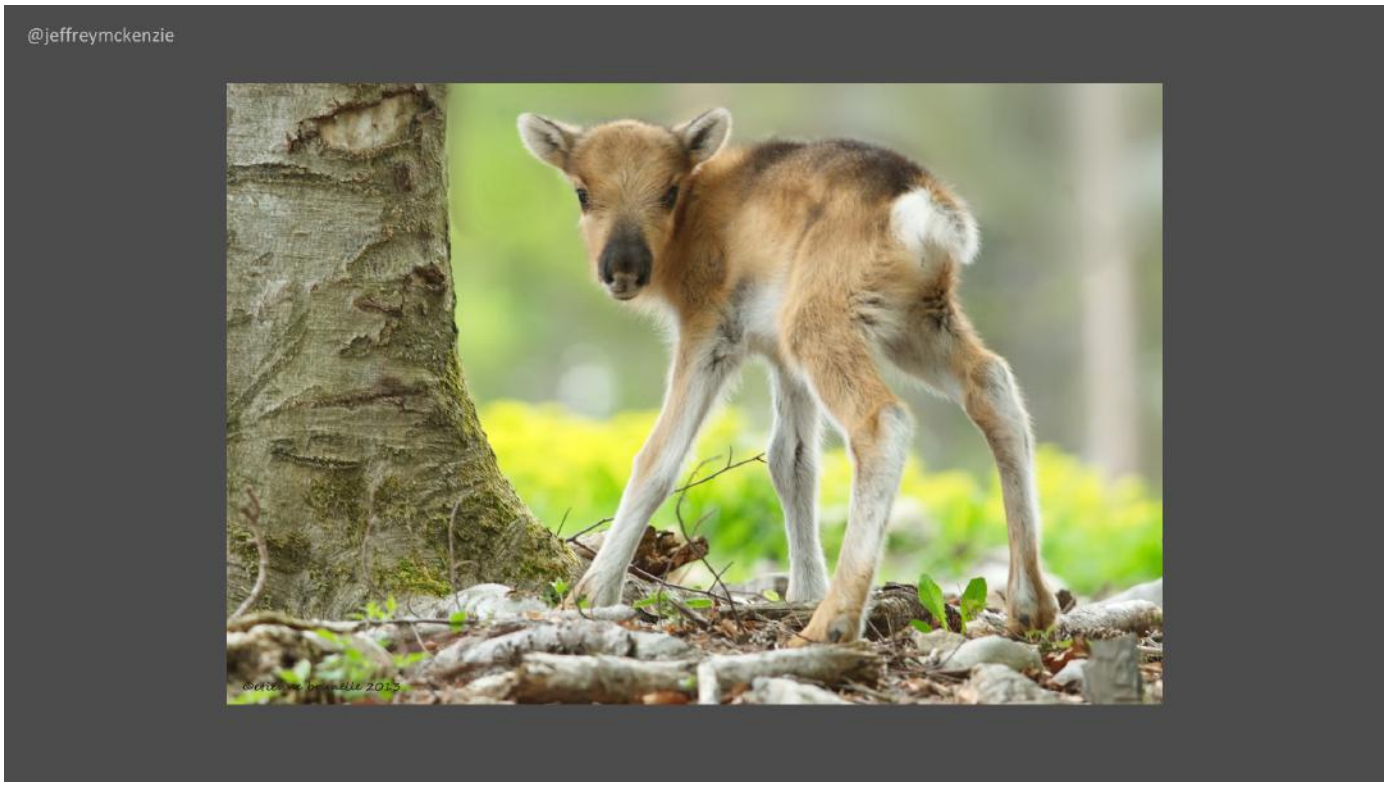
Then it gets changes from the Santa repo…



Then it runs our script
and we get the expected output…



And when we look at the stage view
We can see our variable value (Rudolph)
Was used to name it.



So that's better – we have a very small project/job
That uses shared code and is easy to maintain.

However, we still have to create all those jobs.

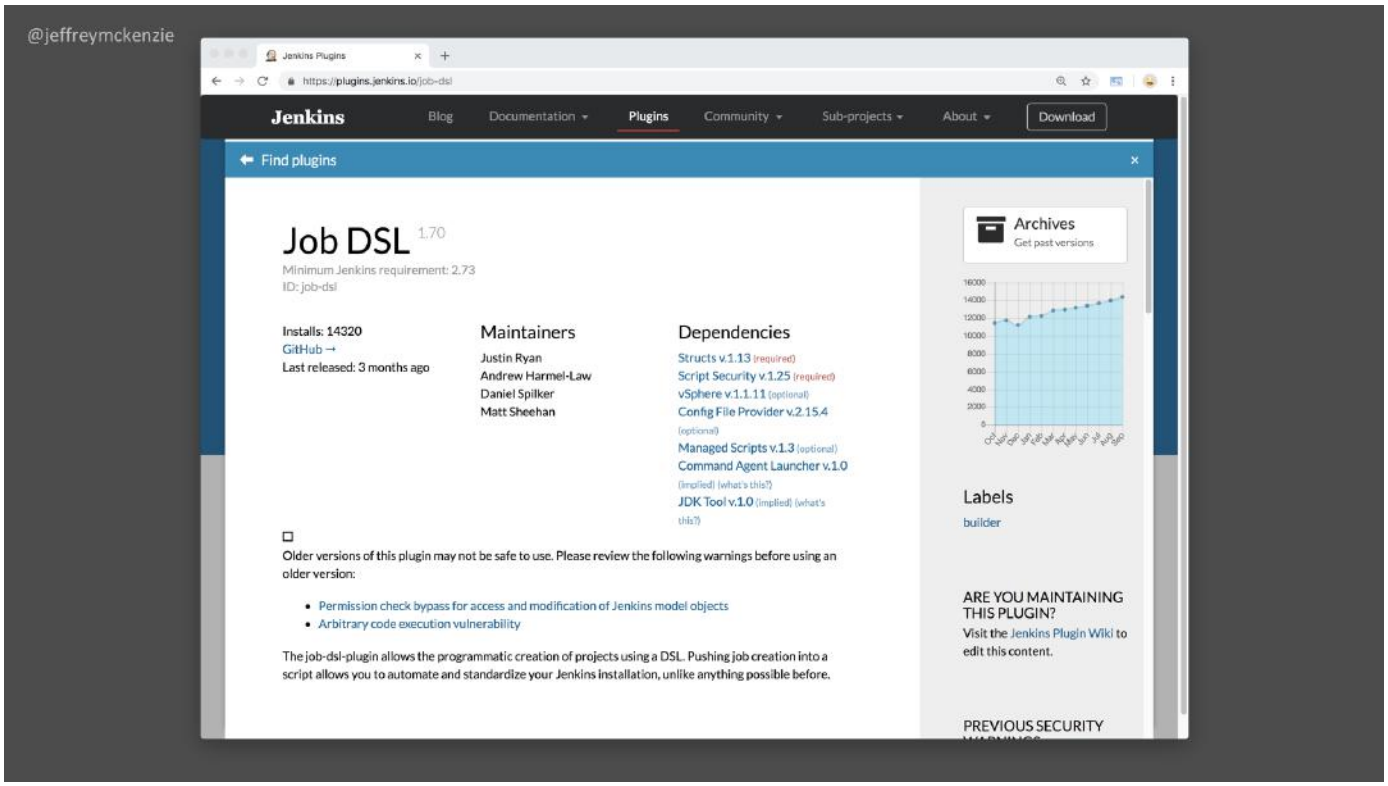
Now Santa's a busy man…
==========
https://commons.wikimedia.org/wiki/File:HanRenne.JPG
By GrottesdeHan [CC BY-SA 3.0 (https://creativecommons.org/licenses/by-sa/3.0)], from Wikimedia Commons

He and Mrs claus have a lot to do around Christmas,
And he really doesn't want to have create all those individual jobs.

He's already automated his communication with his reindeer –

Wouldn't it be nice if he could automate his automation?

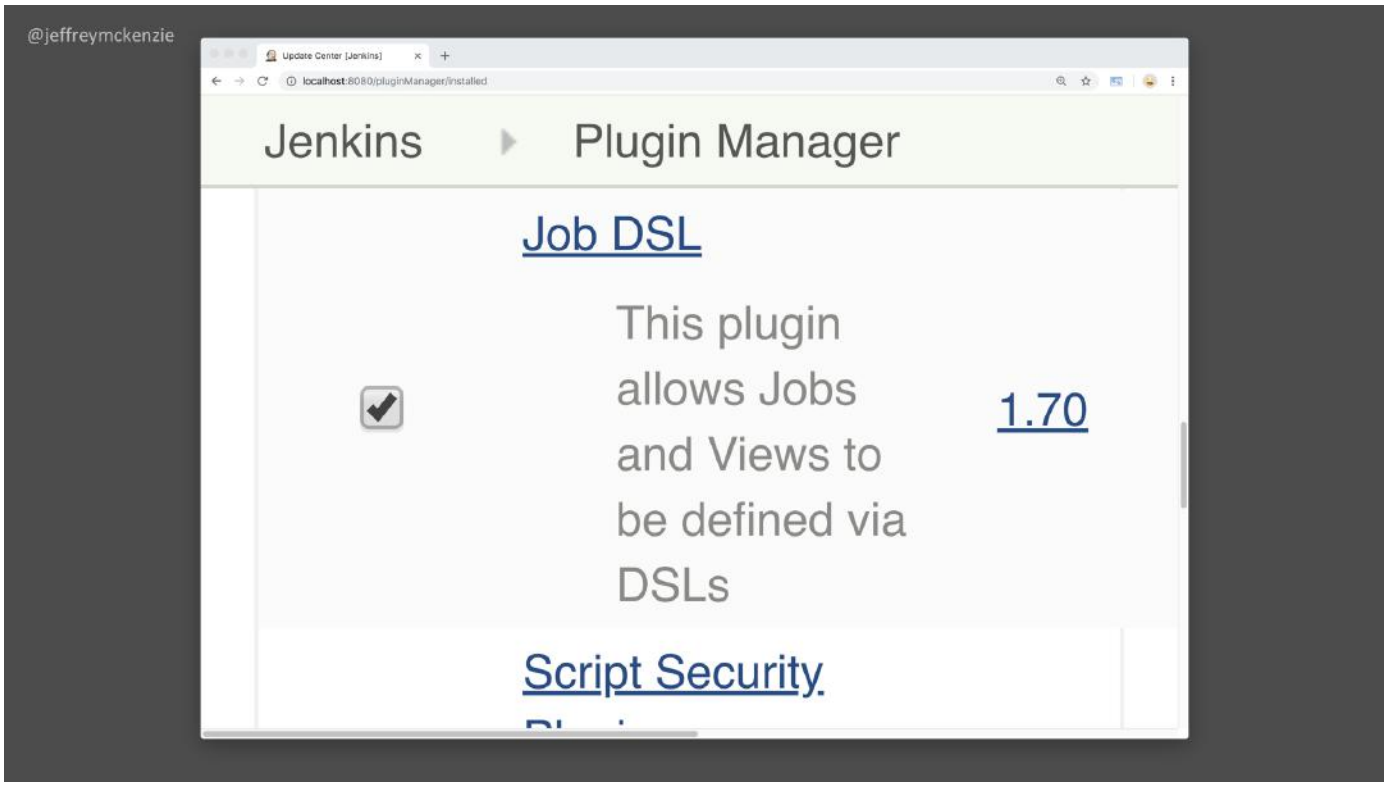
Well it turns out he can…
==========
https://commons.wikimedia.org/wiki/File:Christmas_came_early_for_one_Alaska_village_151016-Z-MW427-433.jpg
By Staff Sgt. Edward Eagerton [Public domain], via Wikimedia Commons



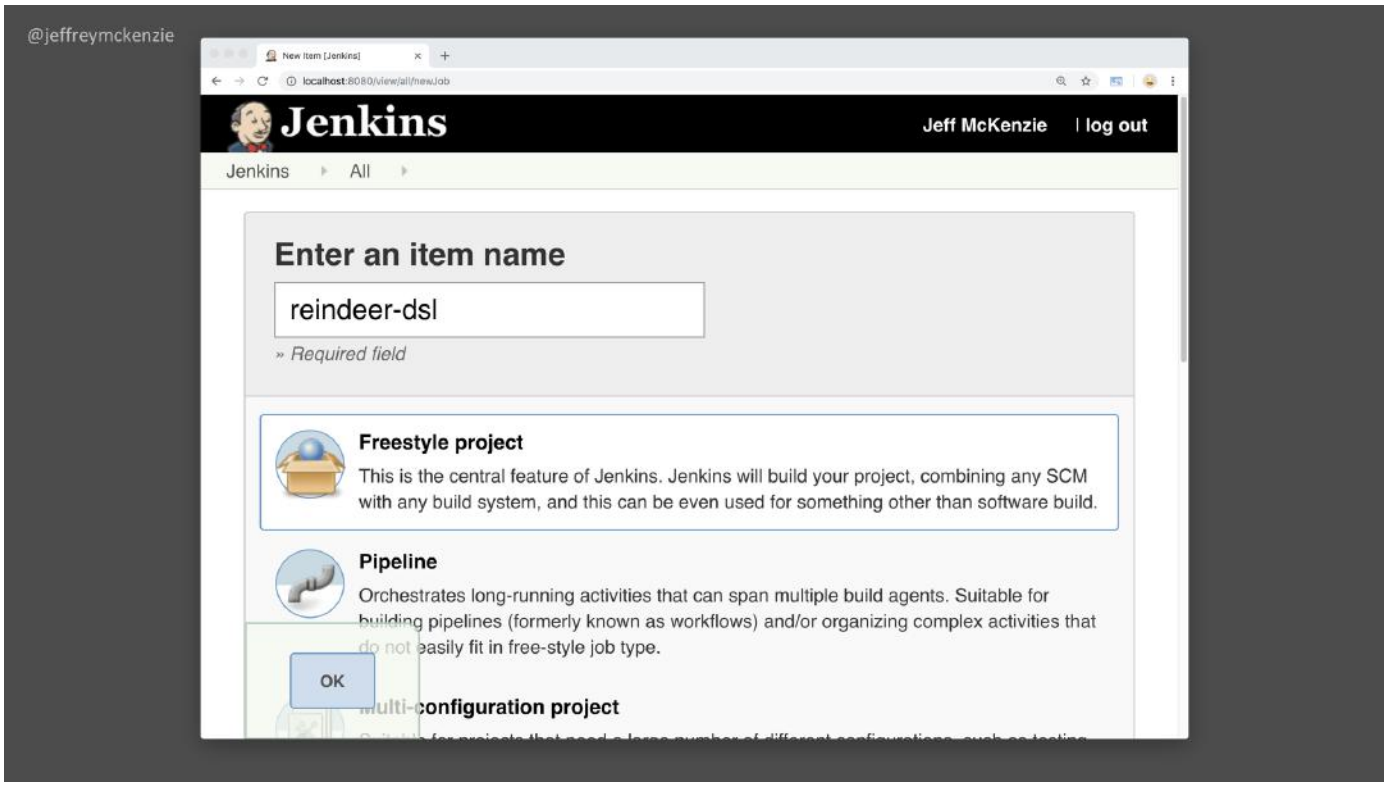
Using the Job DSL plugin for Jenkins.

- a pipeline script allows you to
Programmatically control what a job does,

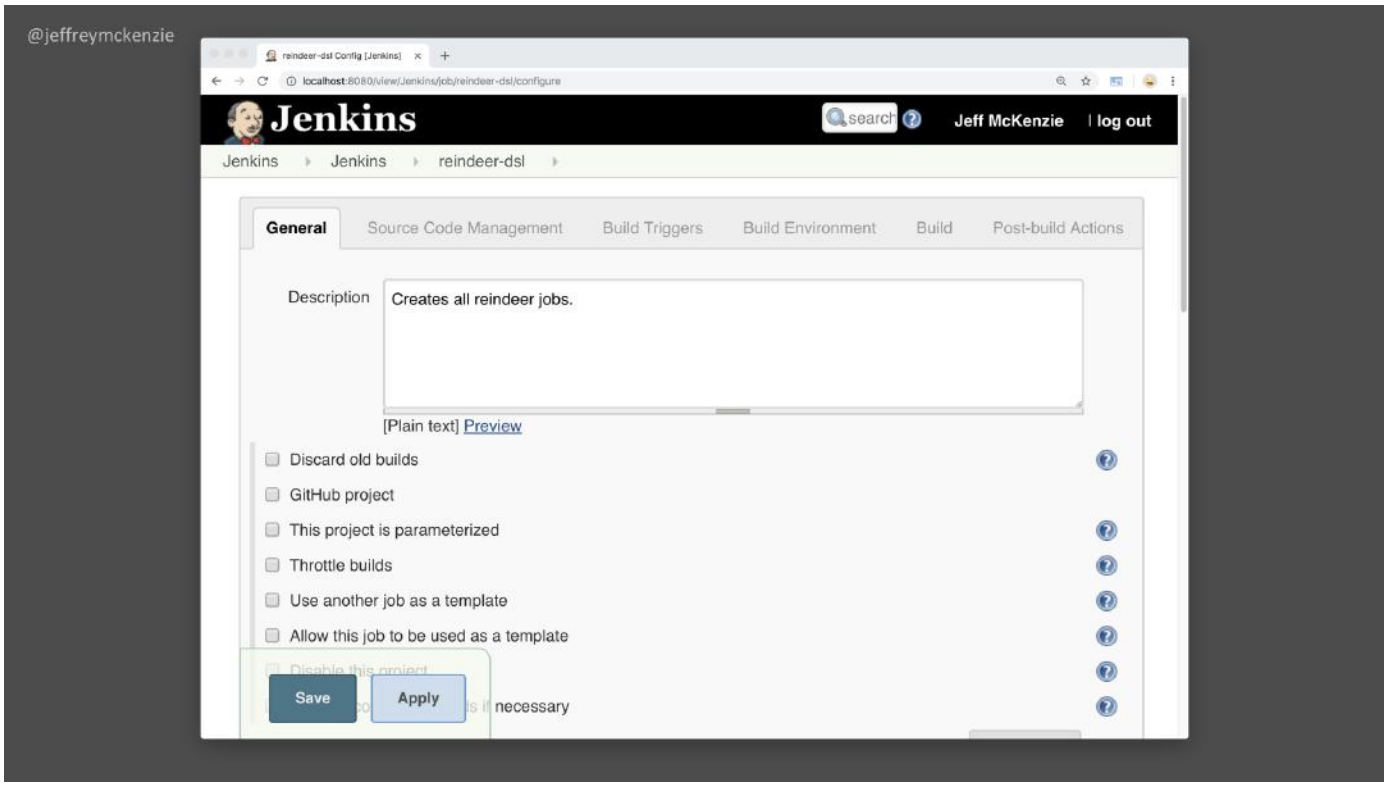
- A job dsl script allows you to
Programmatically create jobs themselves.



This is not built into Jenkins,
But you can install it through plugin Manager.

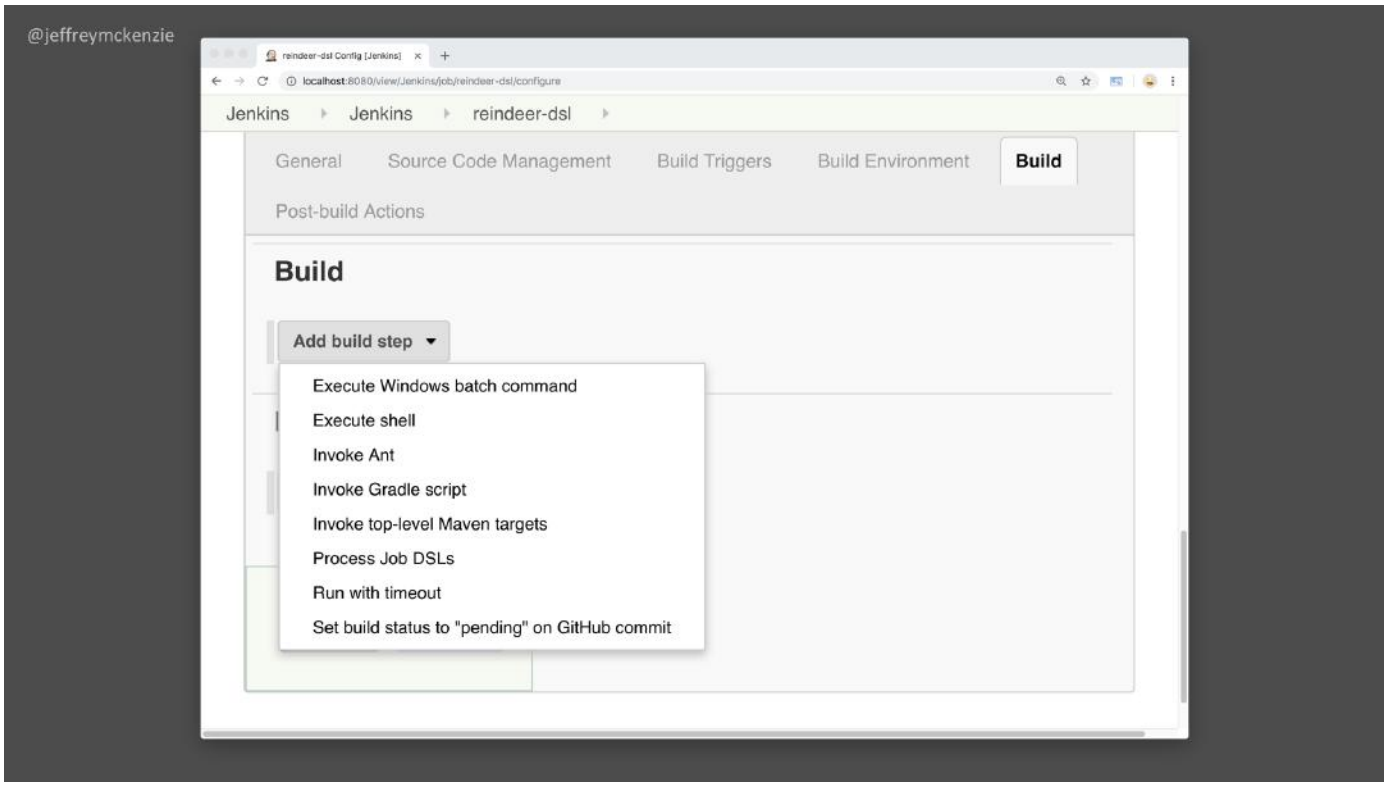


Let’s see how that works --
You create a DSL job as a freestyle project



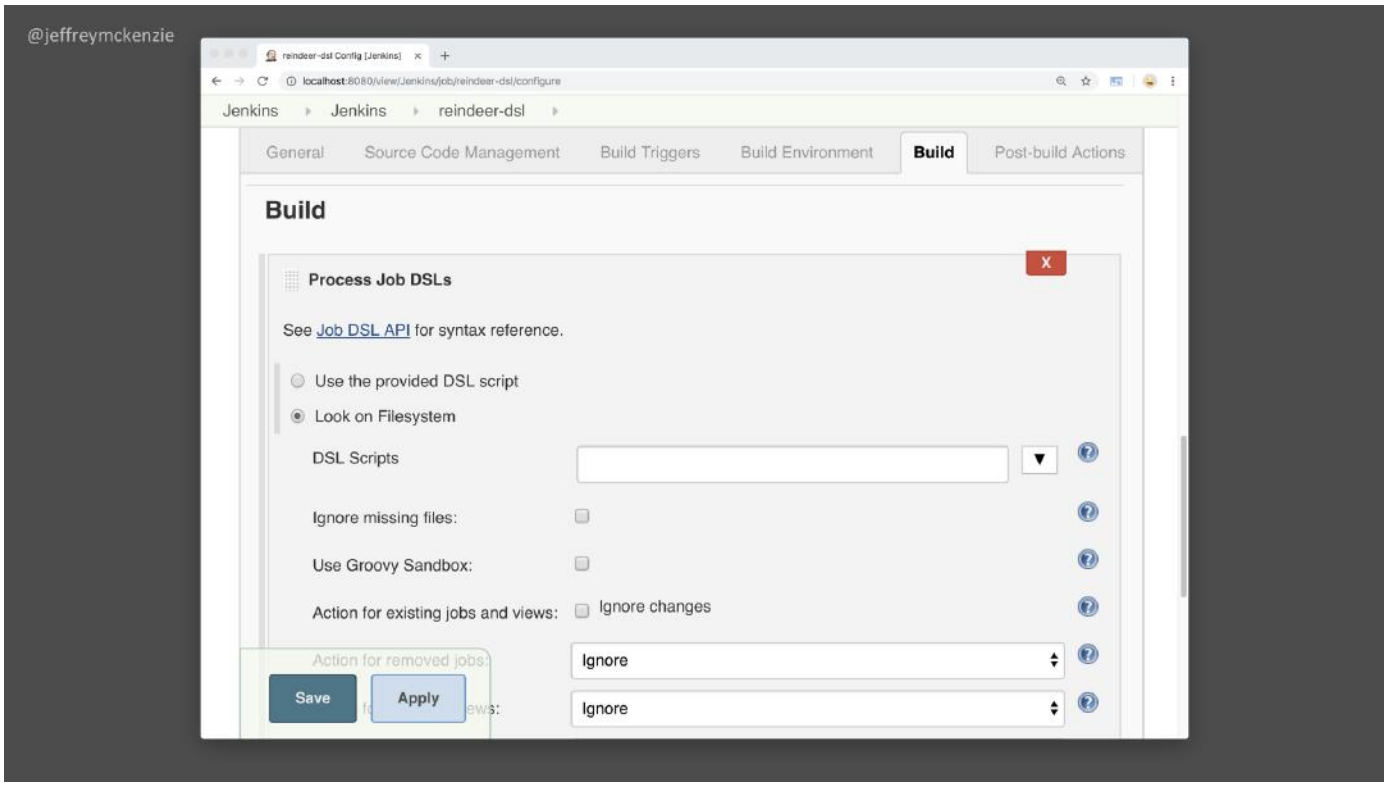
Give description –

Creates all reindeer jobs



Then we go down to build and select the step
Process job DSLs –

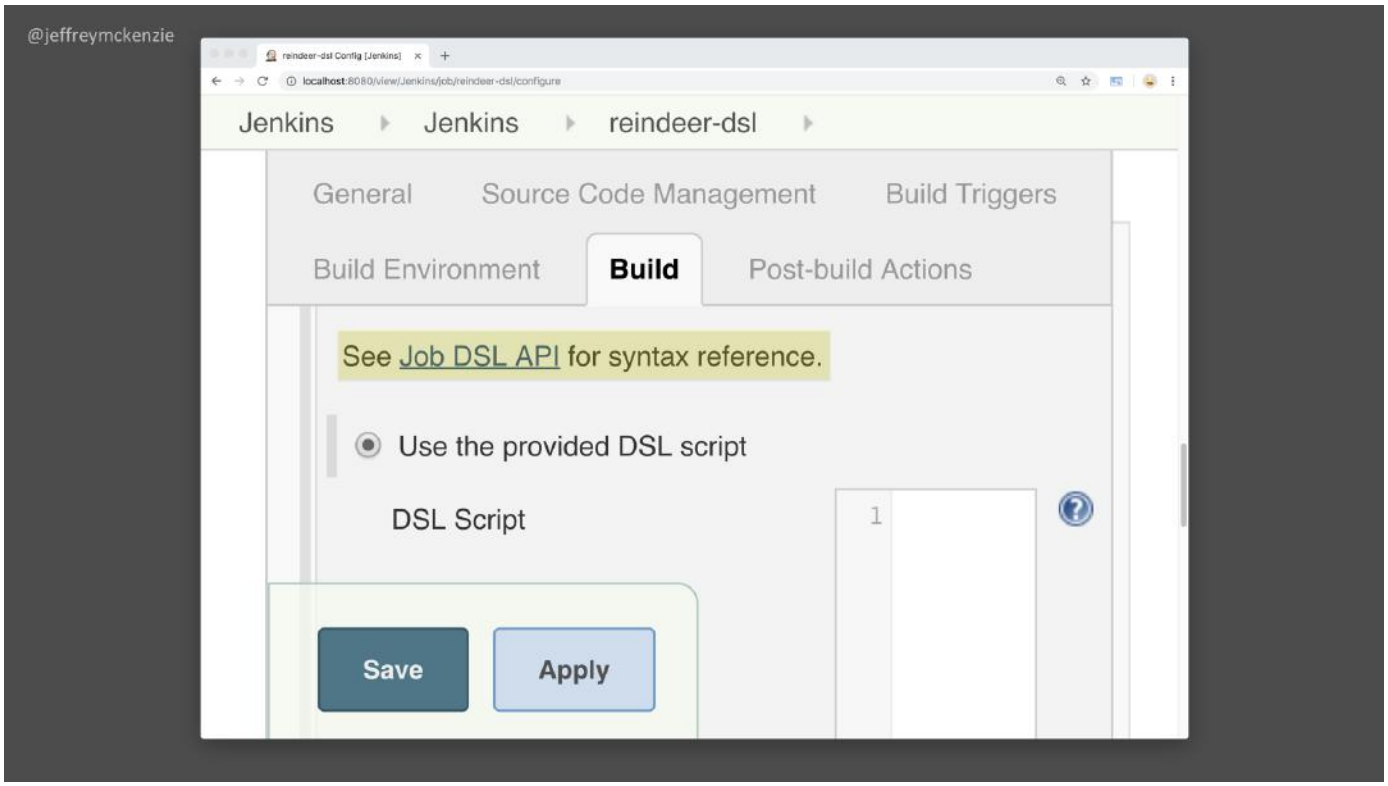
Again, this will only show up
if you have the plugin installed



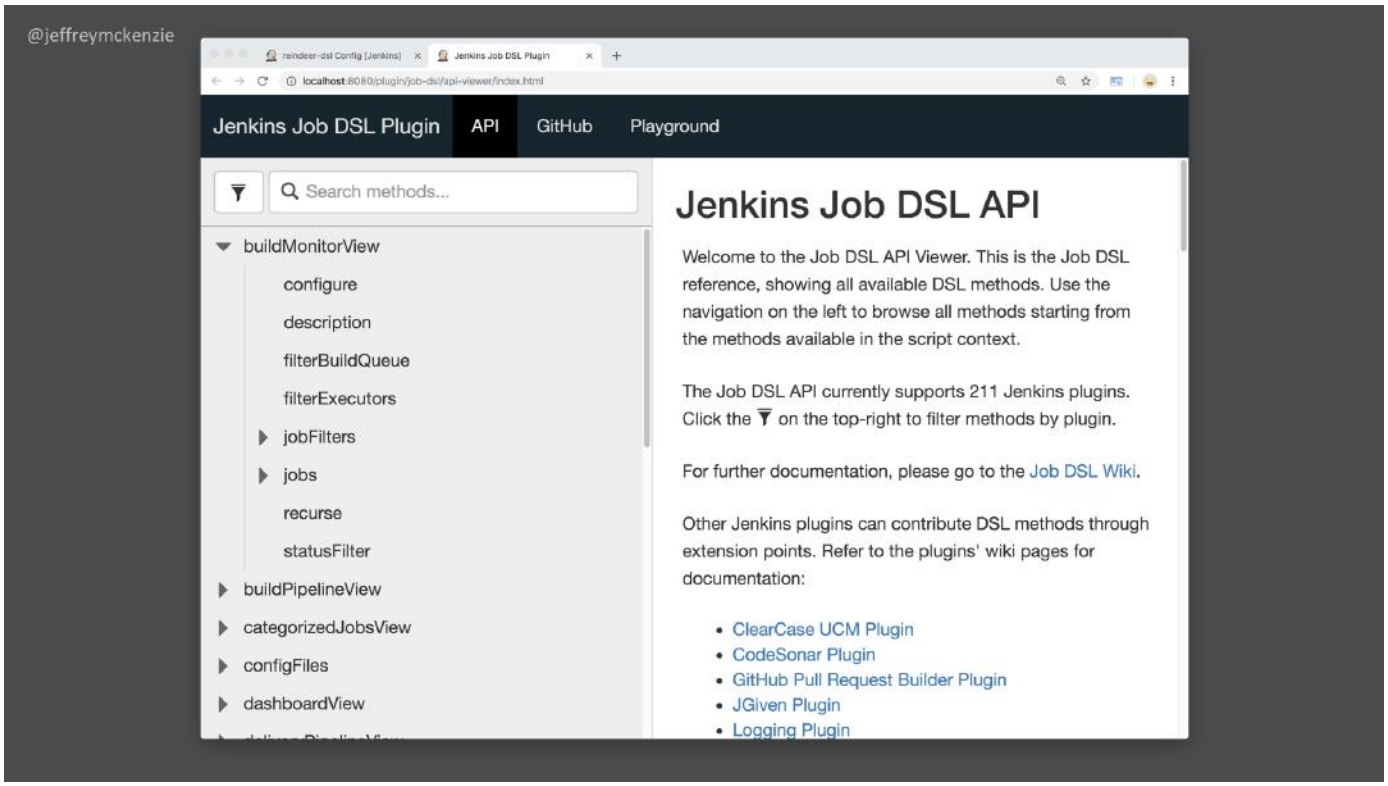
Have a couple options –

We can pull from filesystem,
Which means we can version control this –

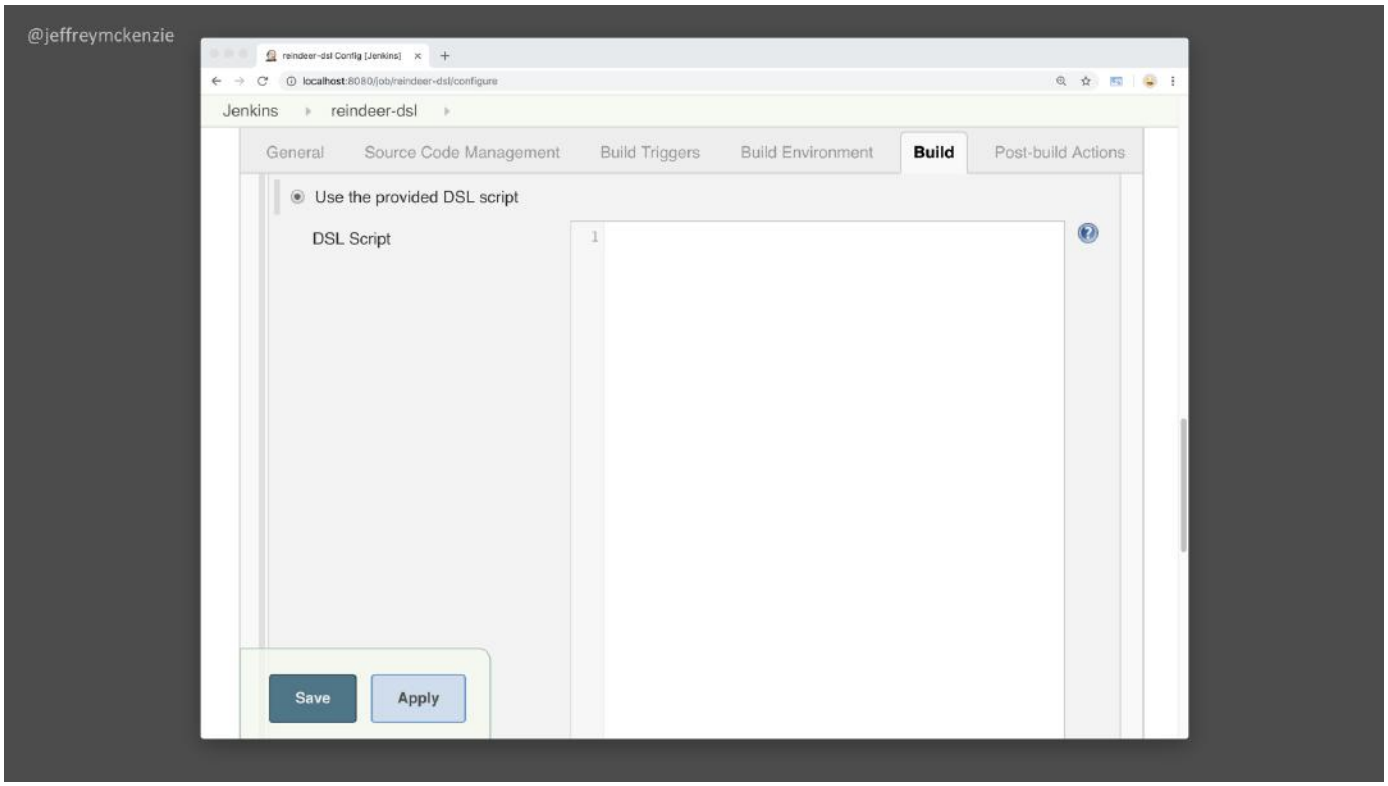
Or we can enter a script right here,
Which we will do for demo purposes.



And on that page there's a link to the
Job DSL API that you can visit
To learn more about the scripting syntax



It has a really nice viewer
That you can search
To find out more about how
To create Jenkins items
Through the DSL



So here’s the space to put our script
Before we do this,
We need to make one adjustment to our pipeline script….

@jeffreymckenzie

```
#!groovy
@Library('jenkins-lib') _

options = [
    greeting: 'Hello',
    reindeer: 'Rudolph'
]
reindeer(options)
```

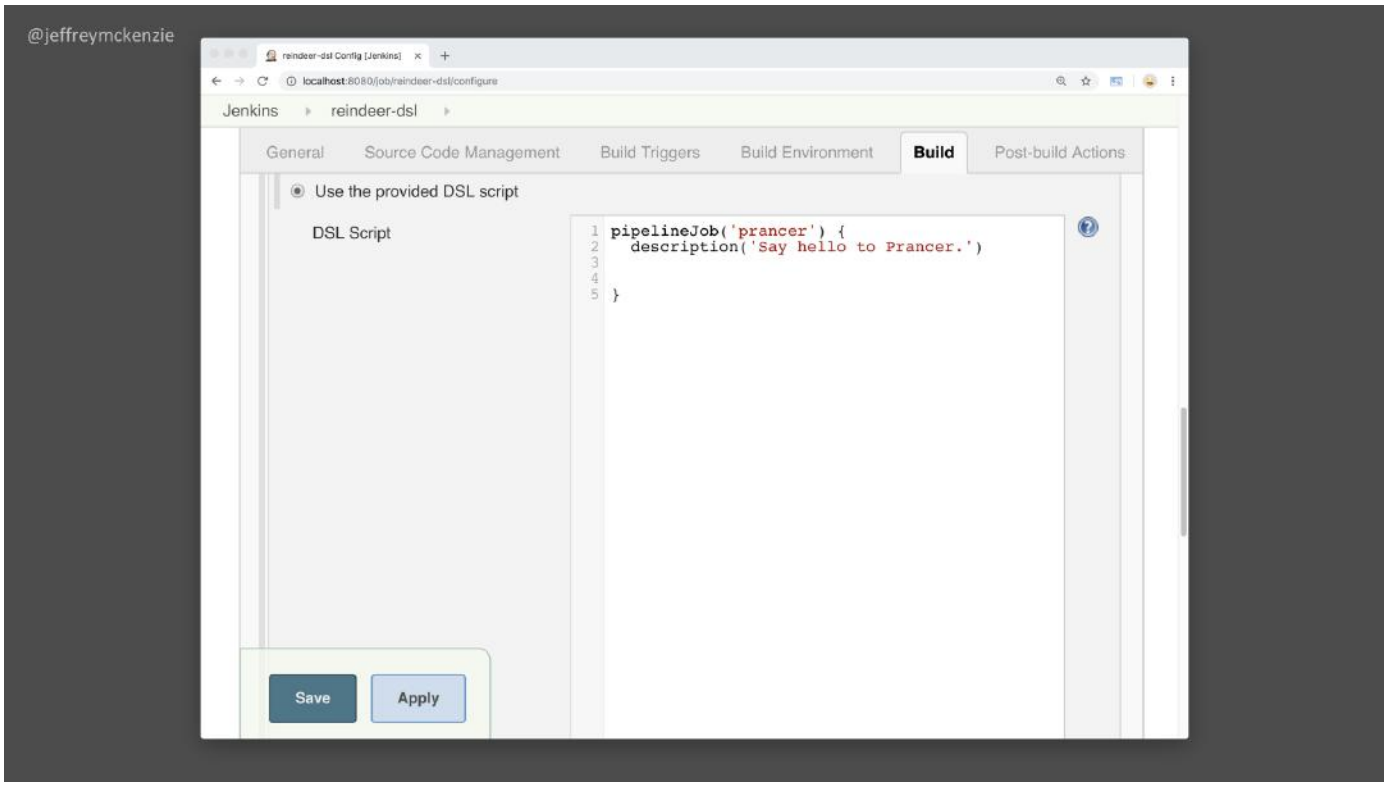
Going to do 2 things –

1. Put this in source control under the name “Jenkinsfile”
2. Switch out the hard-coded values to use parameters

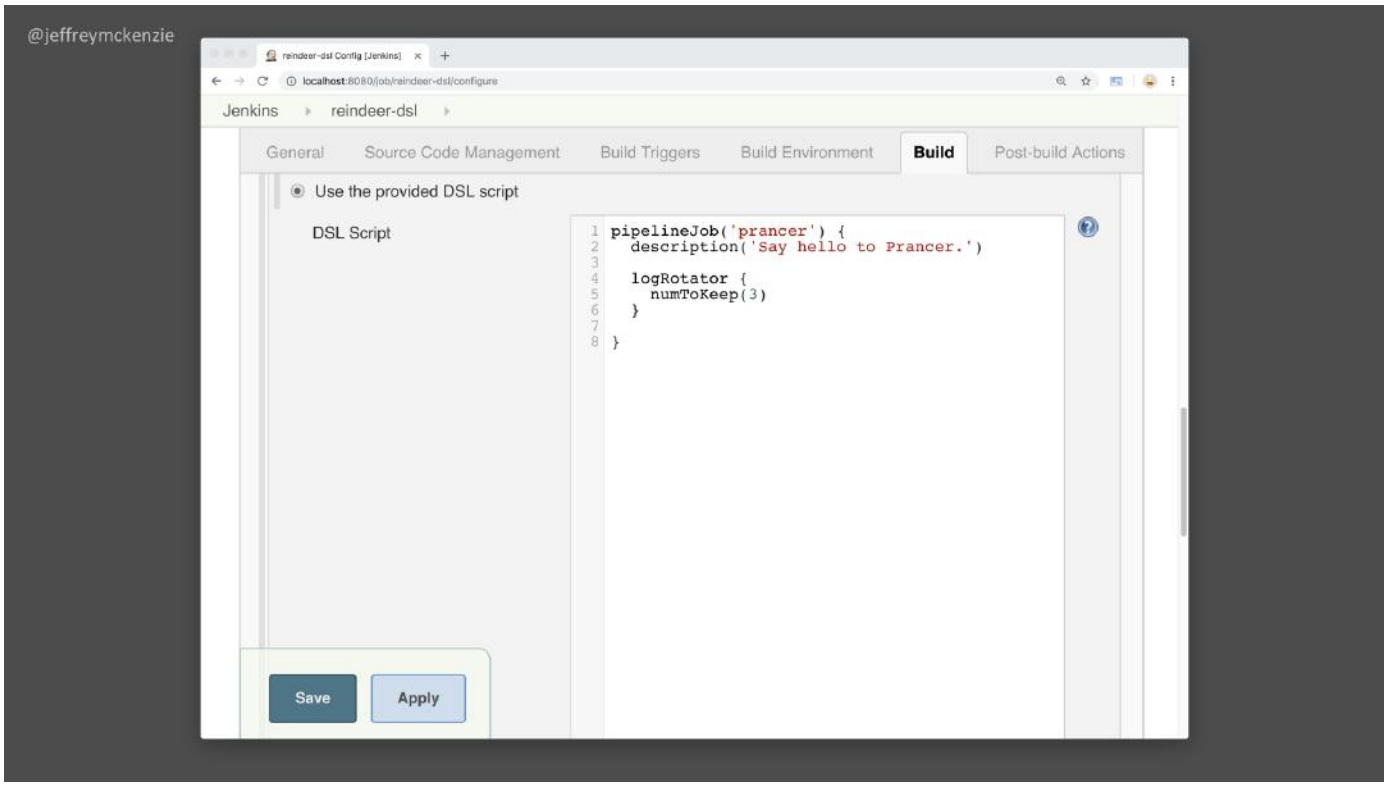
@jeffreymckenzie

```
- Jenkinsfile
#!groovy
@Library('jenkins-lib') _

options = [
    greeting: GREETING,
    reindeer: REINDEER
]
reindeer(options)
```

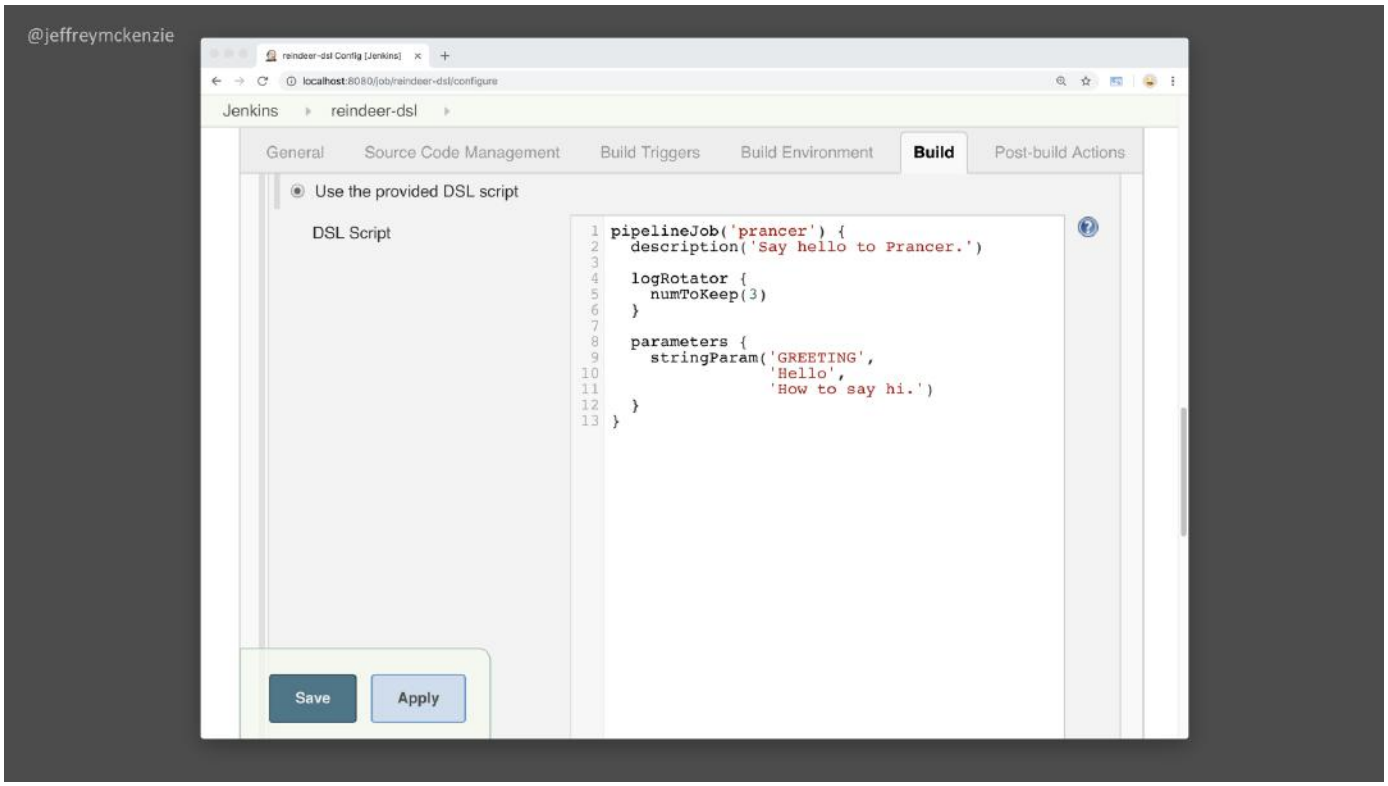
Otherwise, it’s the same.
Lets go back to our dsl project and start to build it



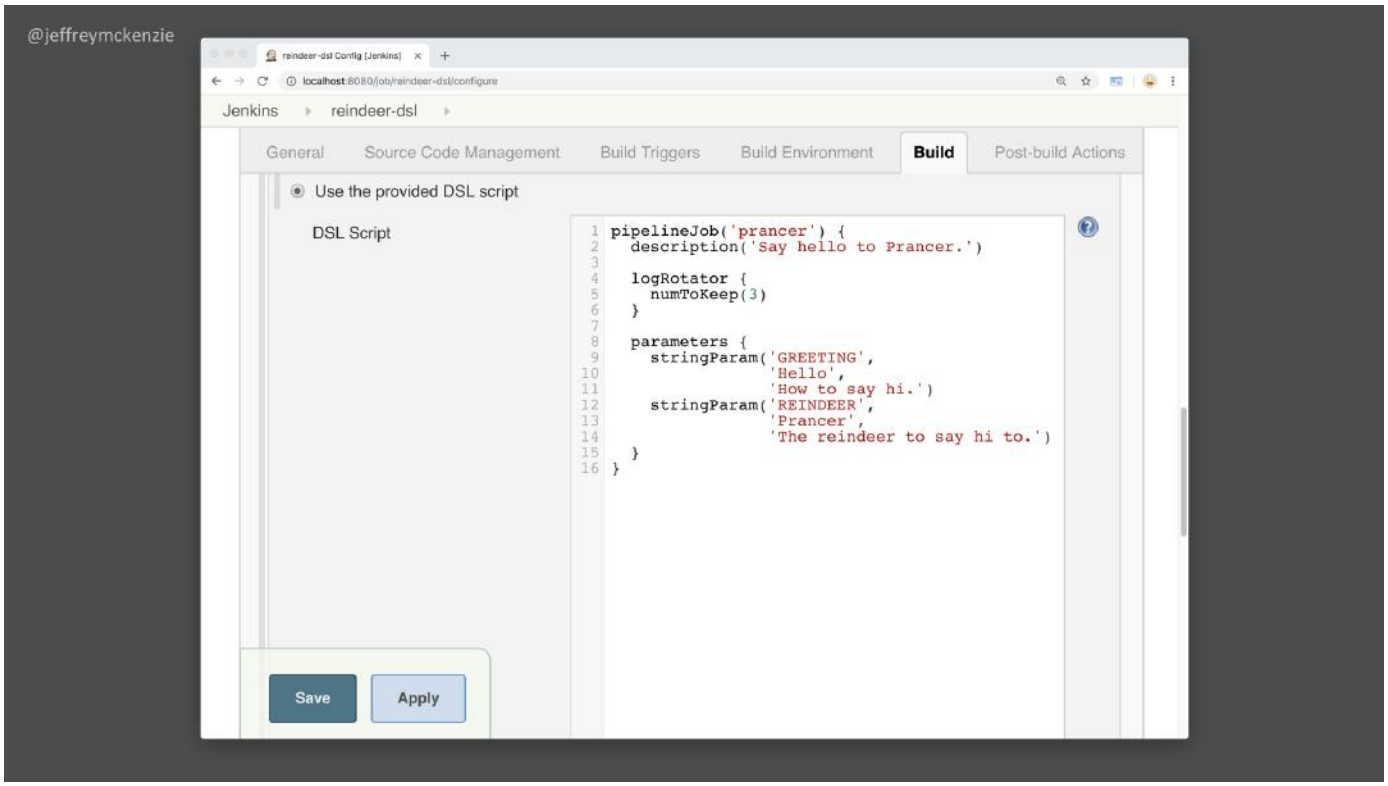
To create a pipeline job or project,
We simply use the pipelineJob command,
With the name of the project –
We'll call this one Prancer.
Then we give it a description.



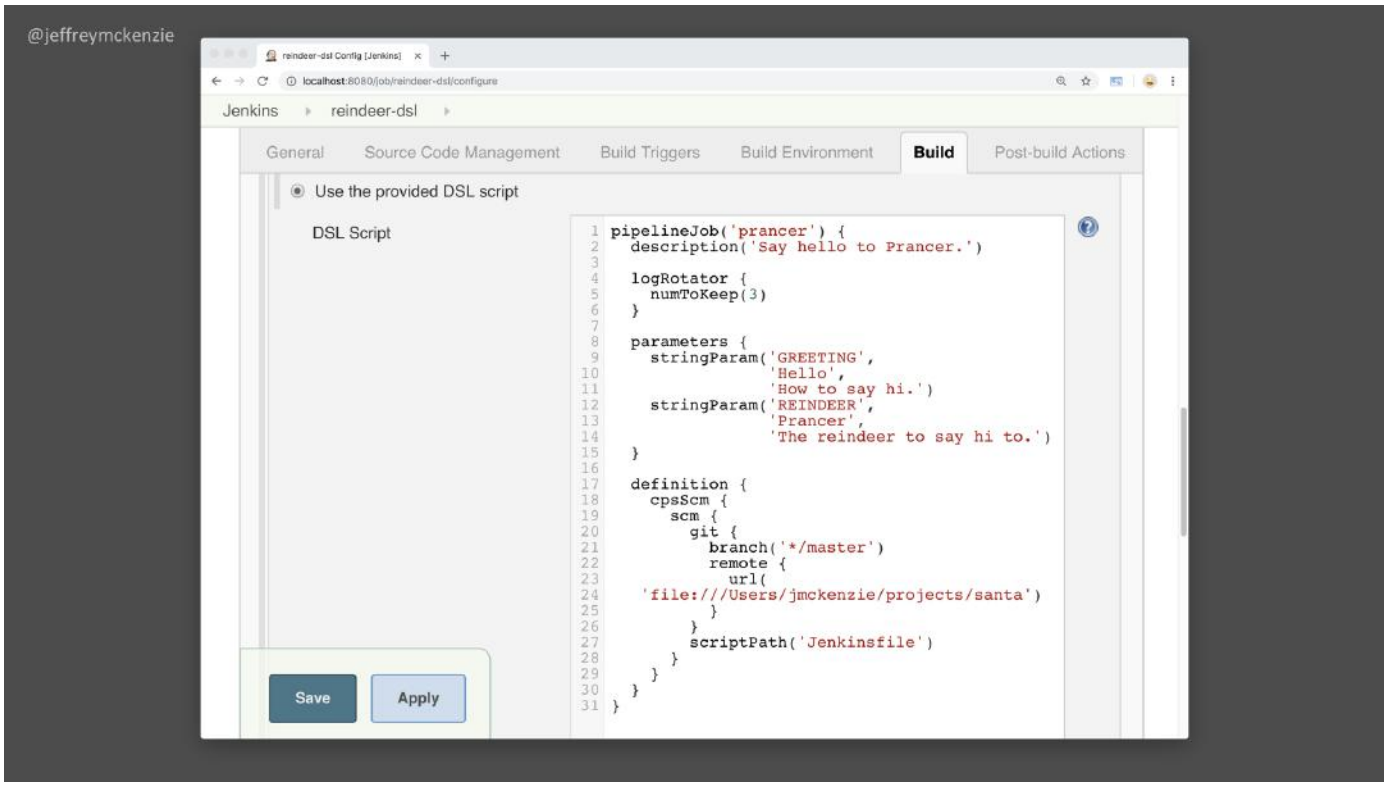
We add our build/log history –
We want to keep the last 3 build logs.



We add our GREETING parameter

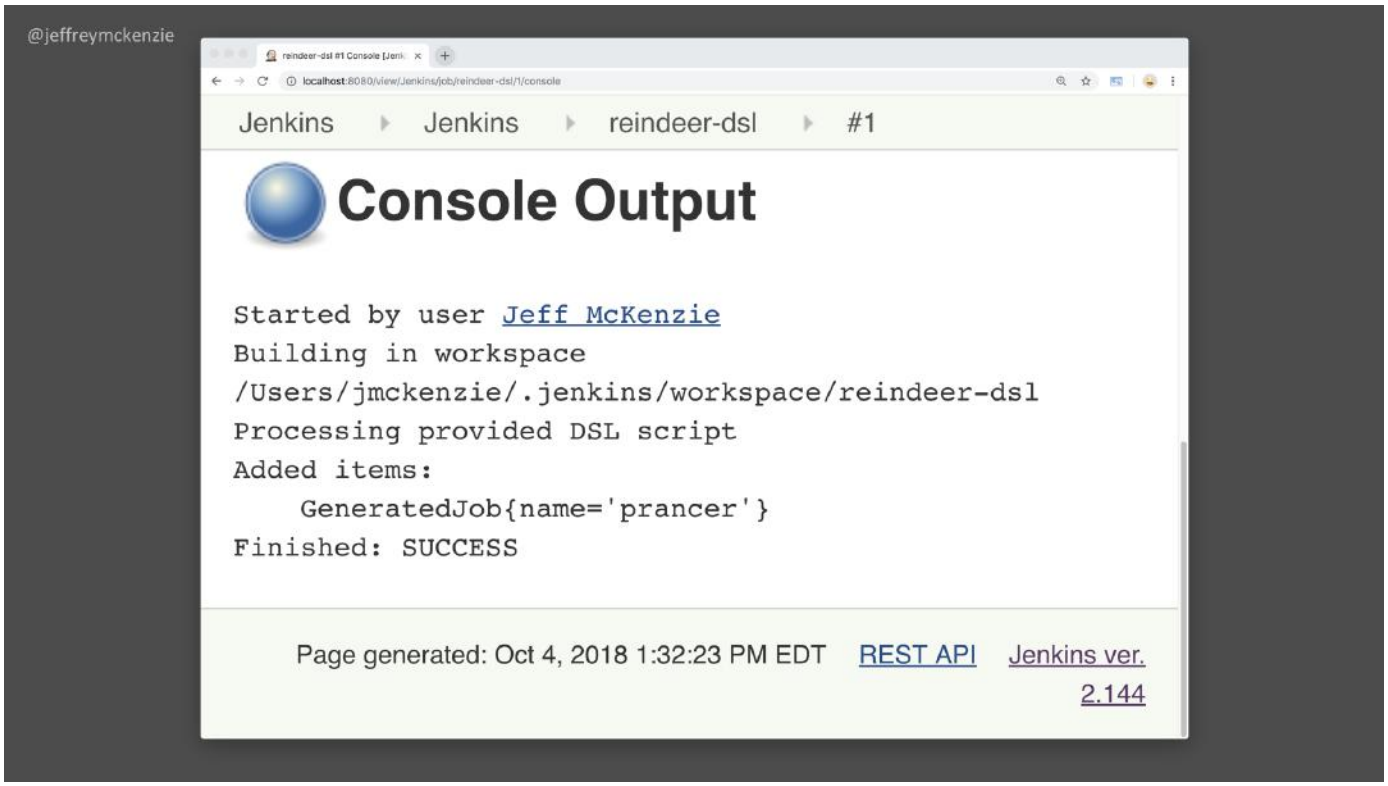


Then our reindeer parameter

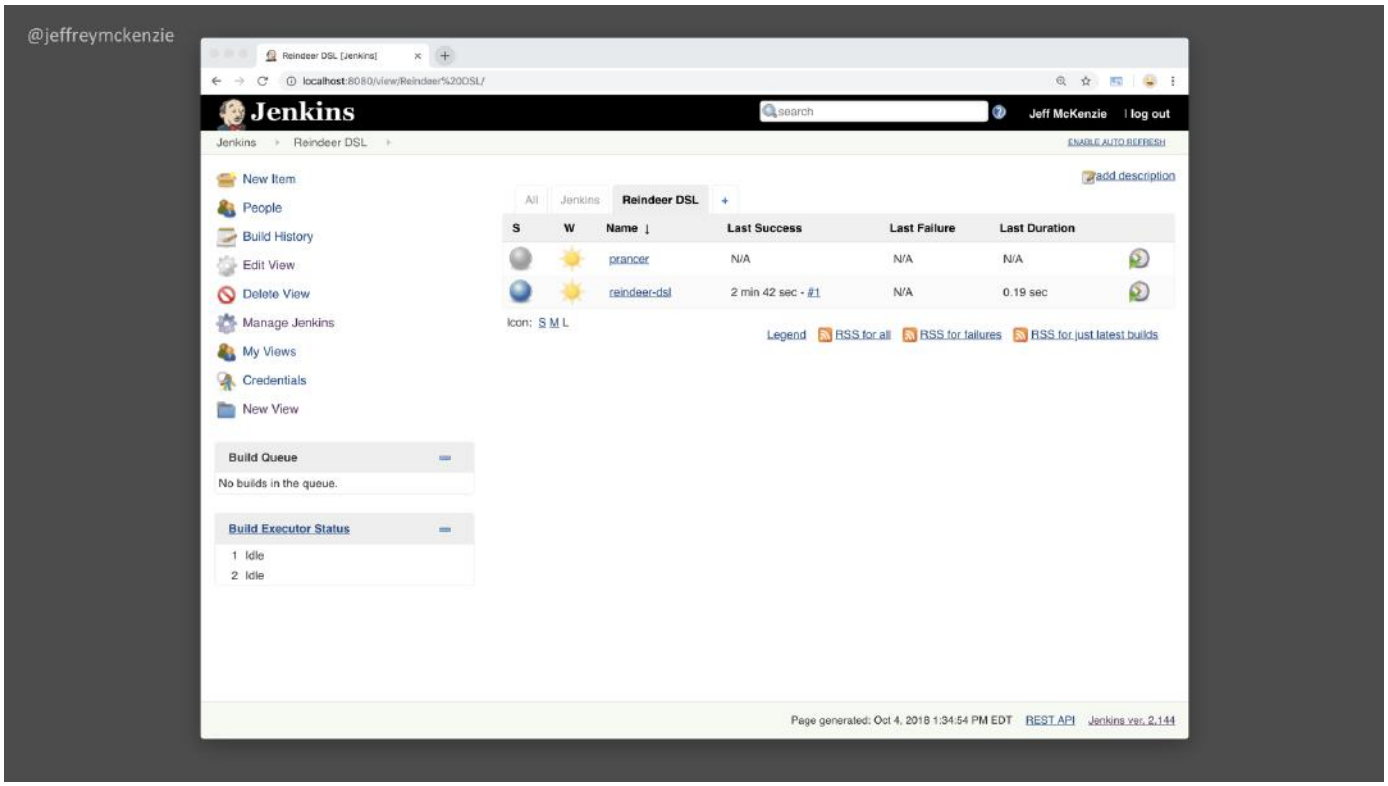


And then finally the build definition itself –
This just tells Jenkins what pipeline script to pull
And which git repo to get it from.

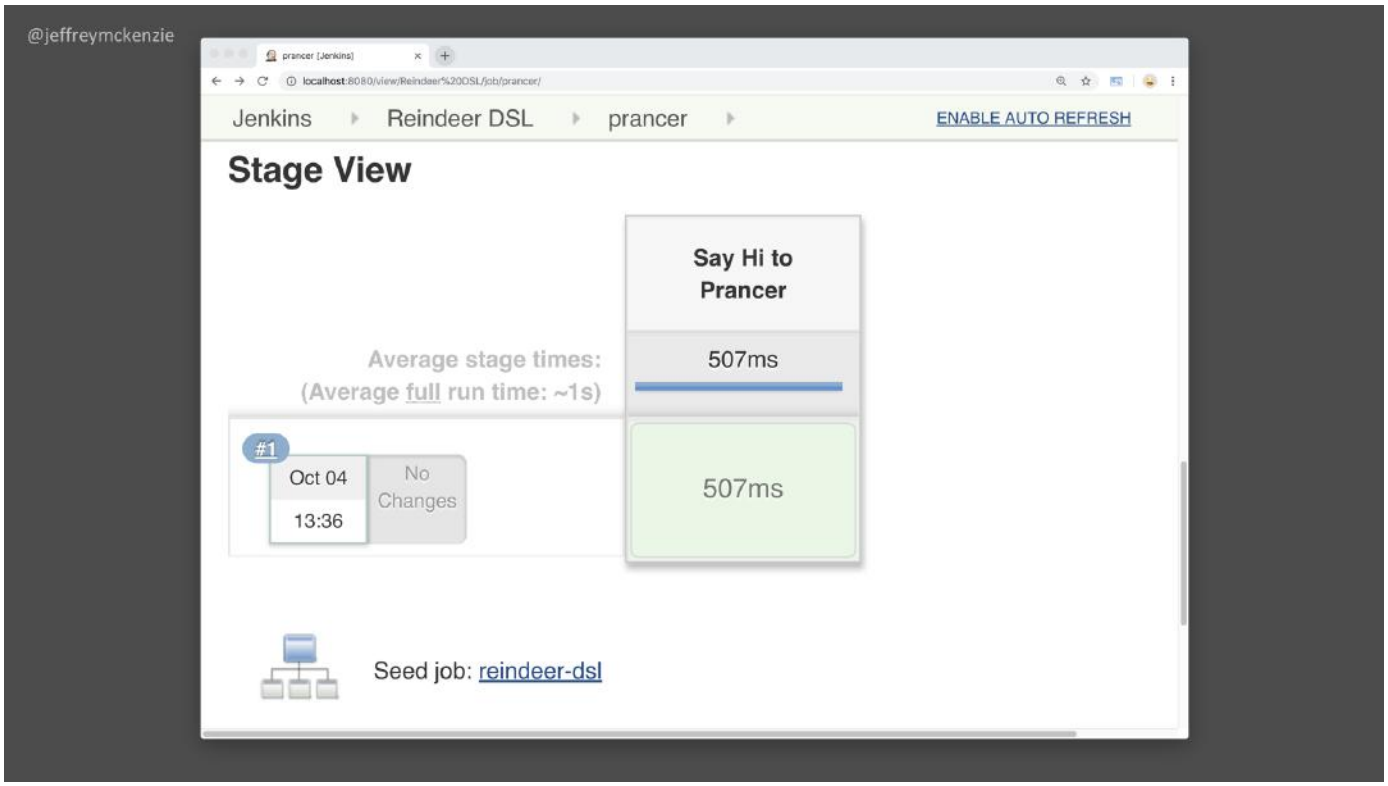
So let’s run this and see what it does…



So it tells us we've created
the prancer job successfully..
Let's go take a look.



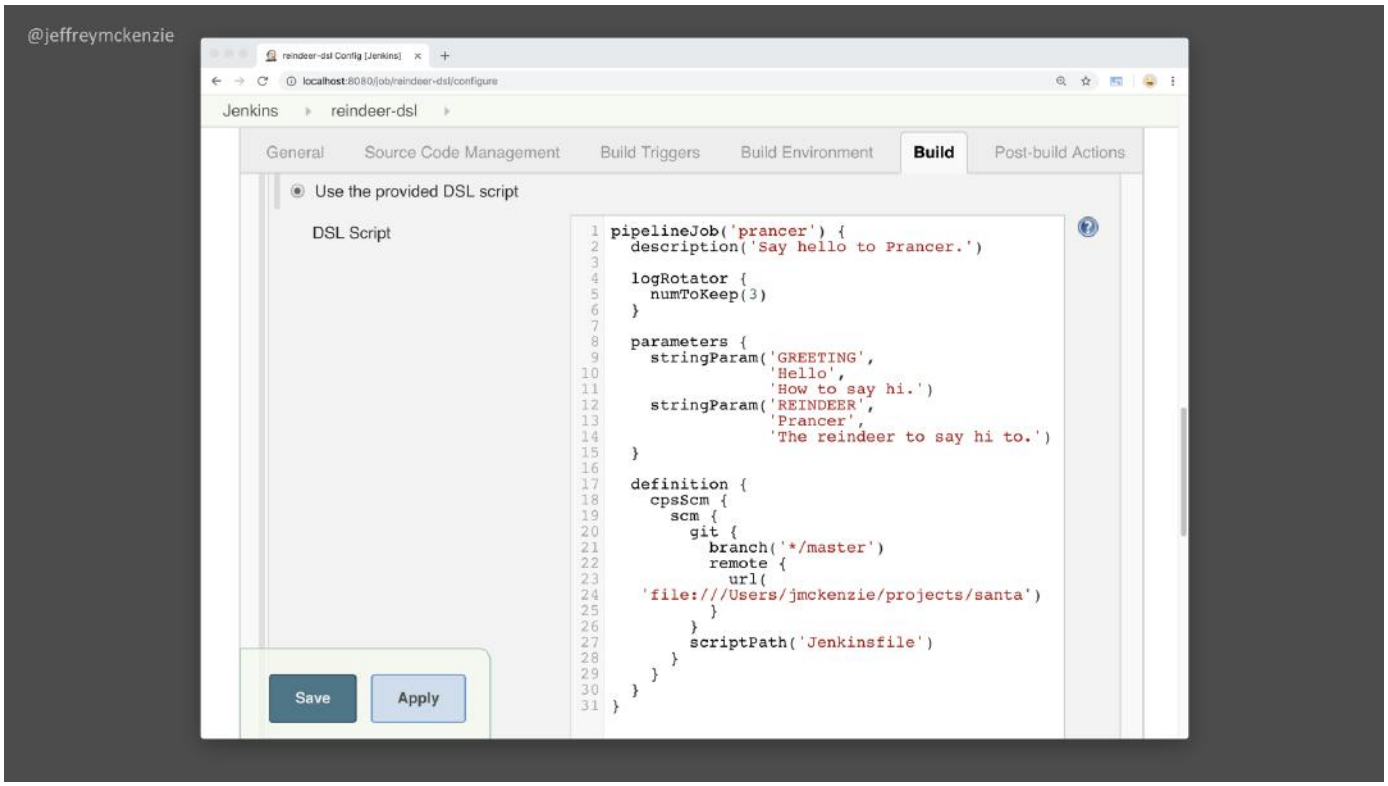
So if we go back to the main list,
You can see the prancer job,
Which was created programmatically
Through the DSL Script.
Let’s run it.



If we look at the stage view,
we can see that it runs successfully,
But this time saying hi to prance

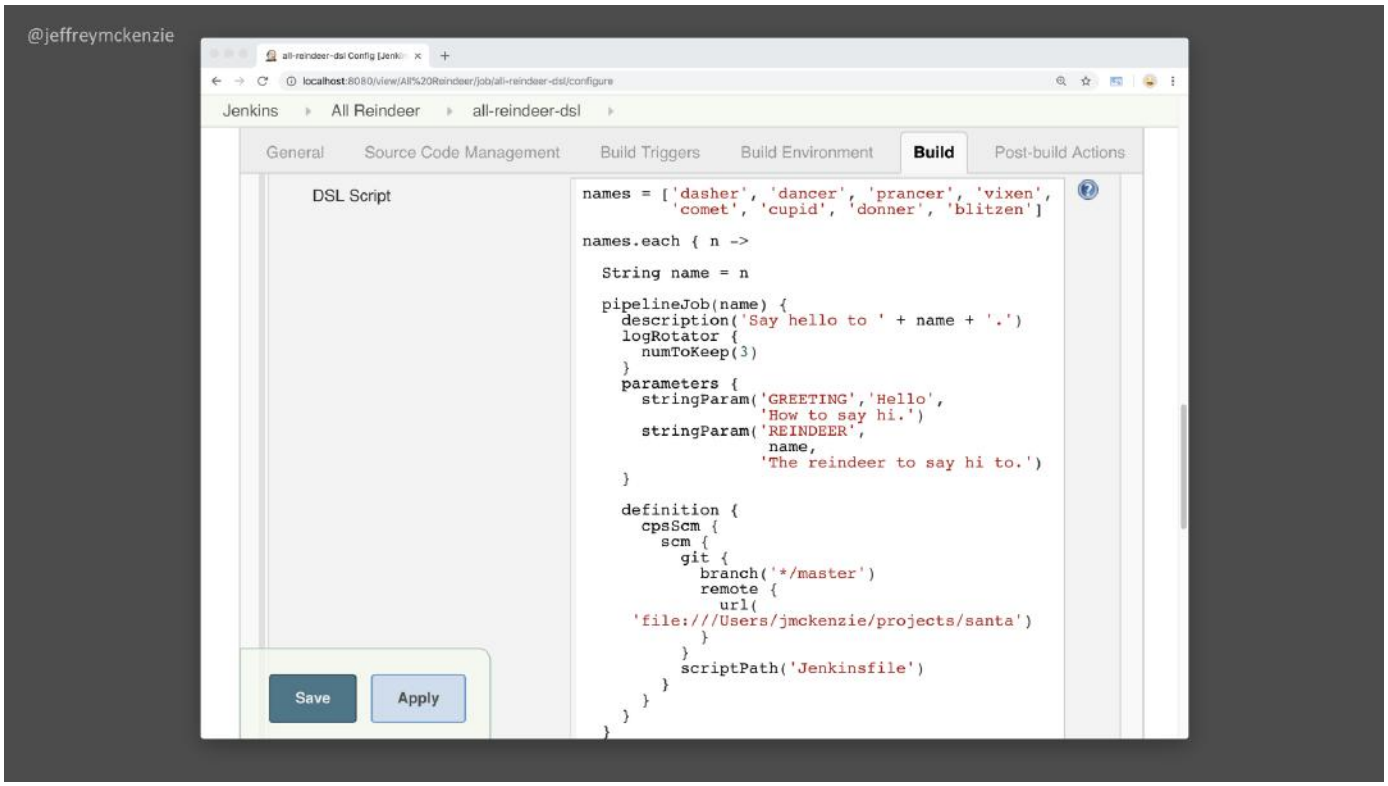
So this is all good, but we've only created one job.

The power of Job DSL is that it allows you
To add scripting to declaration

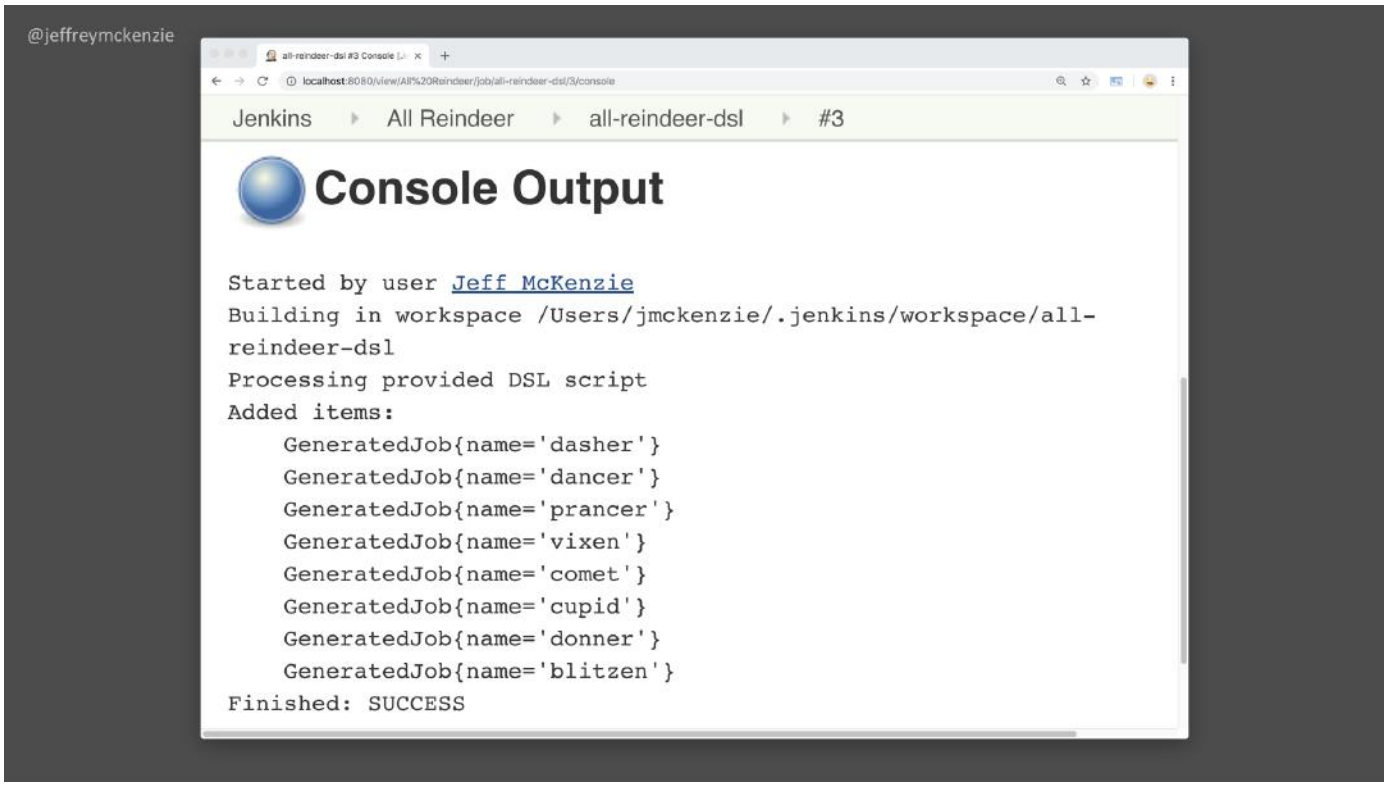


So here's our original DSL script –
Creates the prancer job.

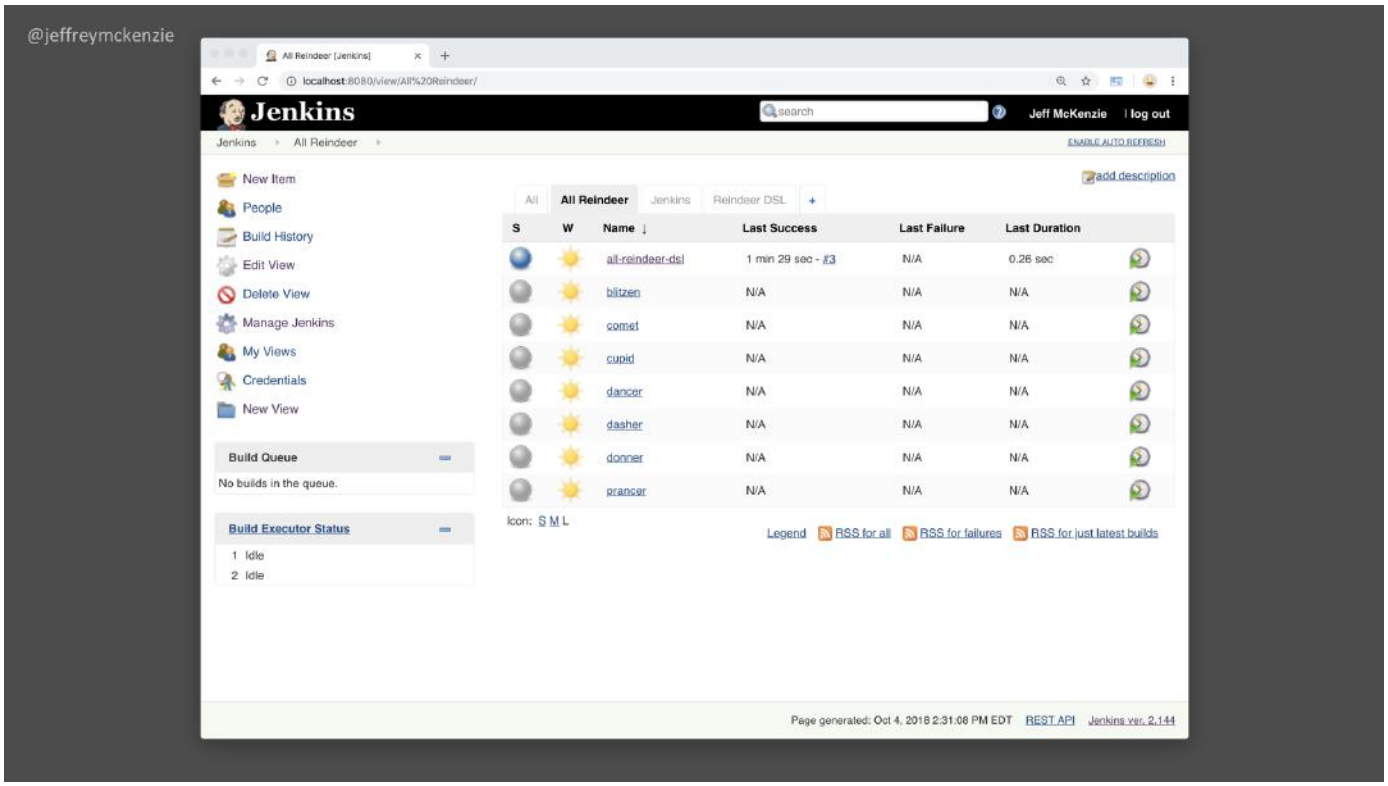
Let's modify that slightly.



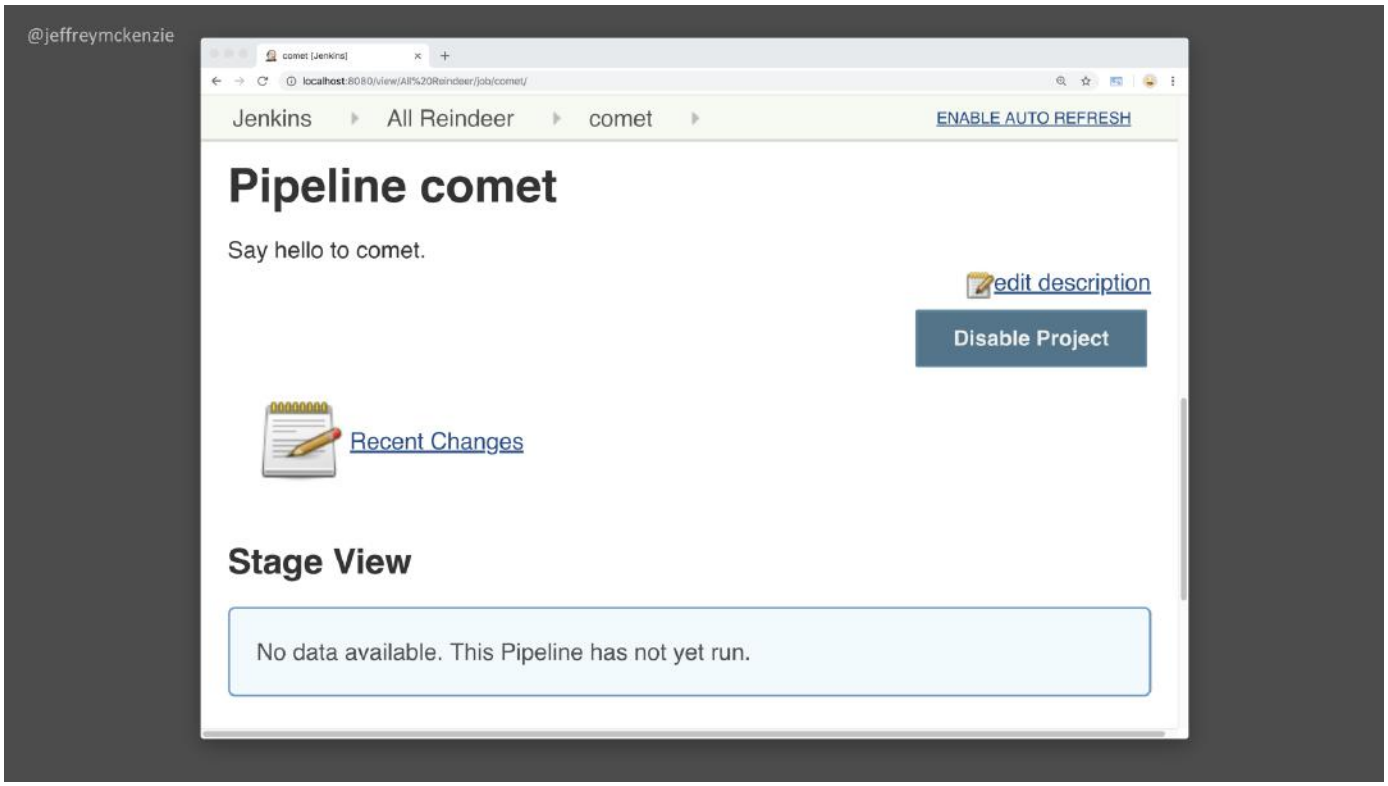
At the top we are going to add an array
Of reindeer names,
Loop through each of those names,
And create a job for each name,
Inserting the name where we need it.
Let’s run this.



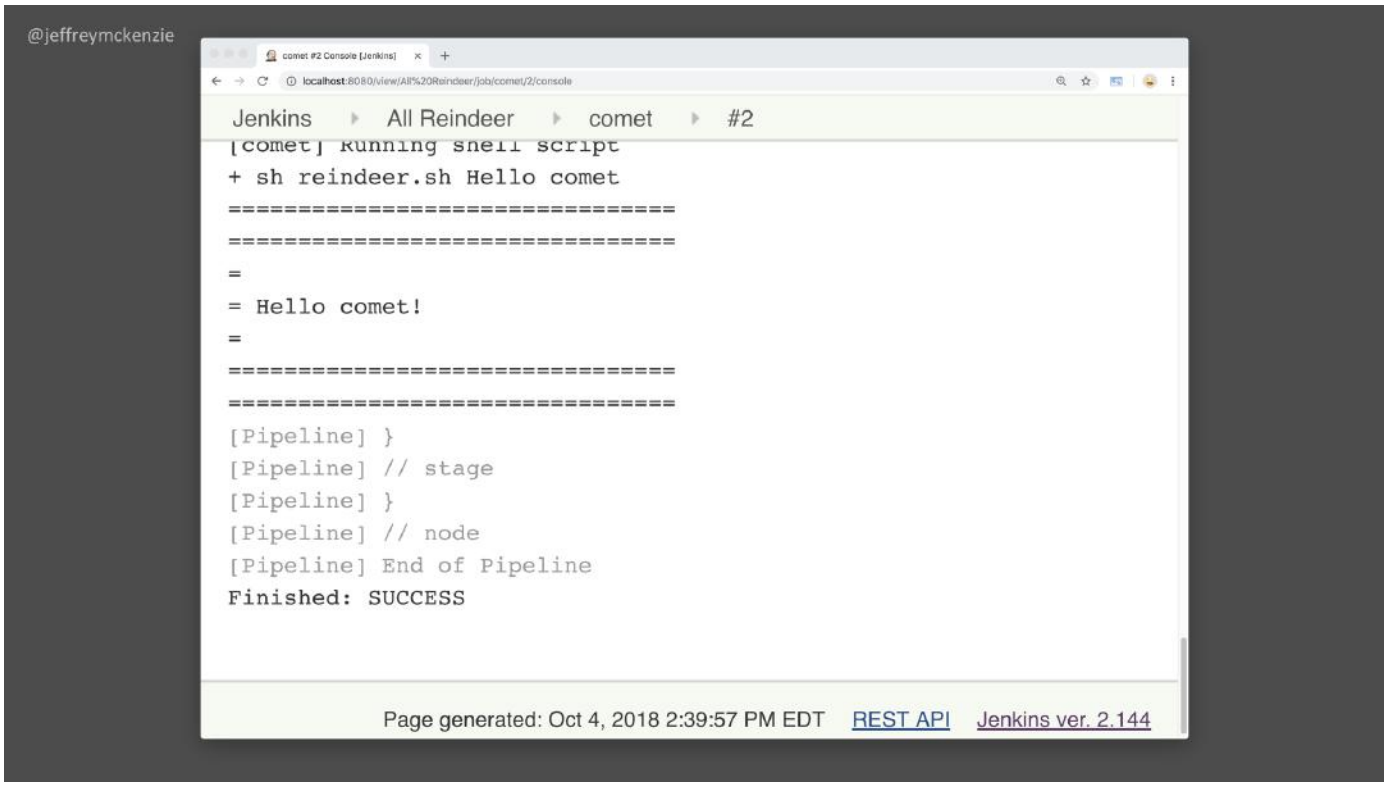
So this created all the jobs –
Let's look at the main page.



There they are.
Now we can drill down into any of these
And run them, and they should
Function just as if we created them manually…

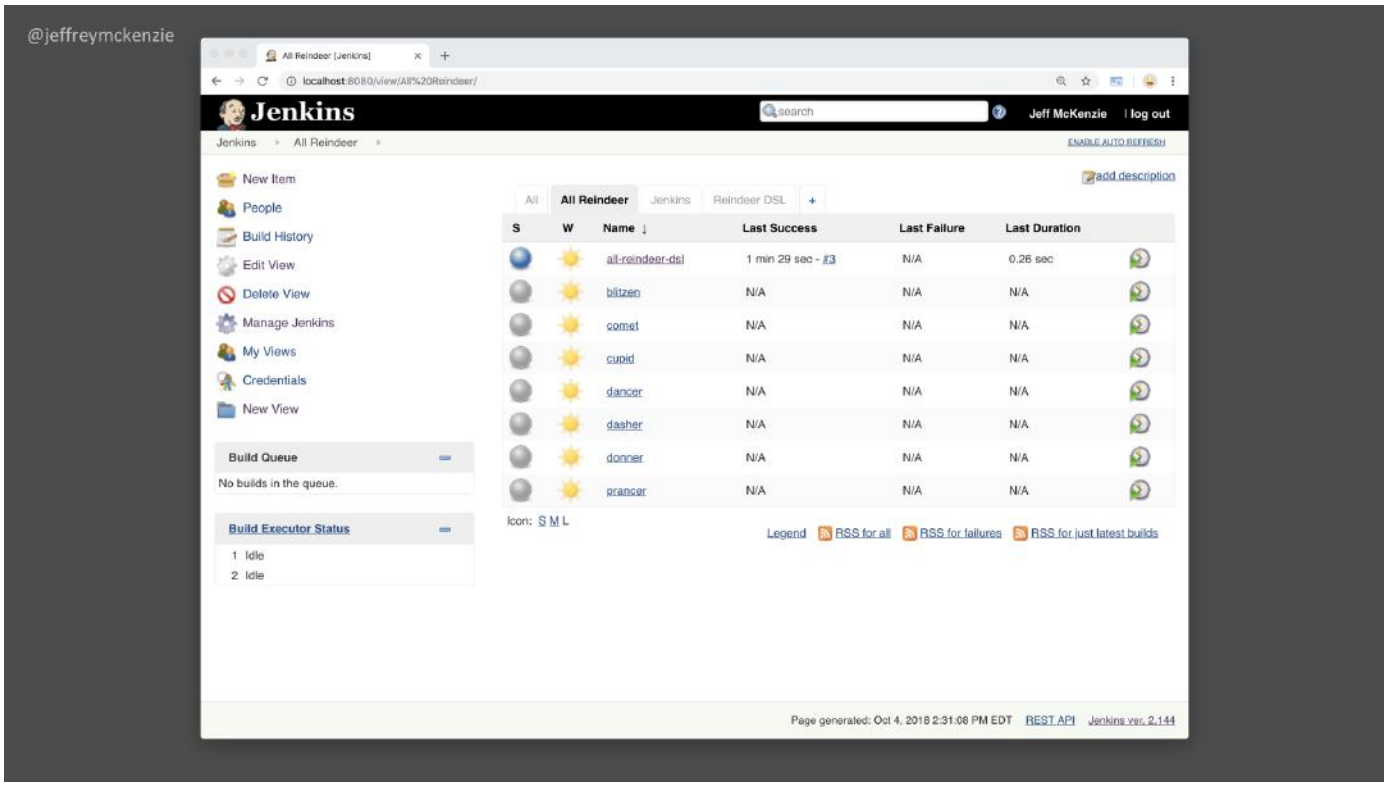


So let's look at comet –
Hasn't run yet…



we run it, and we get the expected result.

So a quick recap of really how little effort it takes
[what santa has been able to do]
To get to this end state…



…of having all these jobs generated.

@jeffreymckenzie

- reindeer.groovy

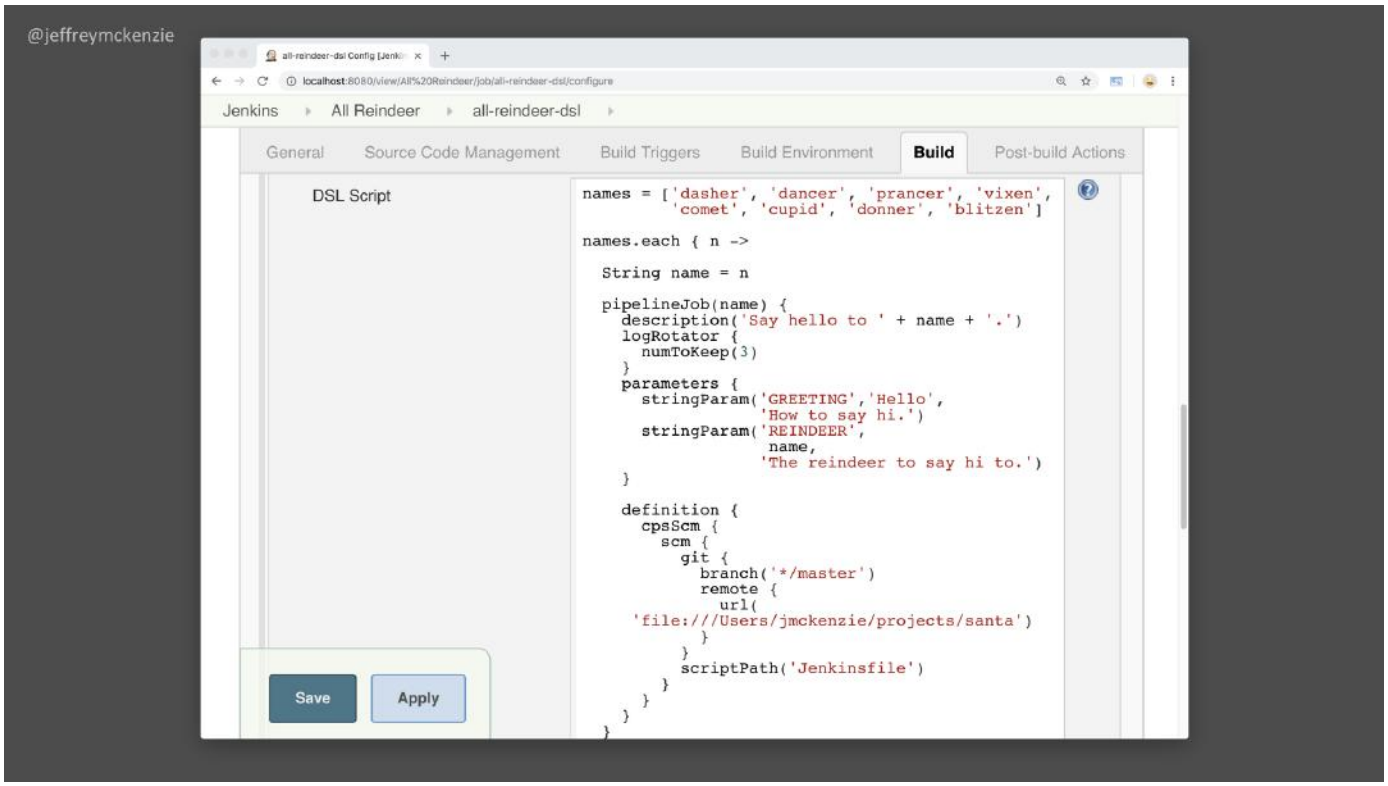
```
#!/usr/bin/env groovy
def call(Map<String, Object> options) {
  try {
    node {
      String greeting = options."greeting".toString()
      String reindeer = options."reindeer".toString()
      stage("Say Hi to ${reindeer}") {
        checkout scm: [
          $class: 'GitSCM',
          branches: [[name: '*/master']],
          userRemoteConfigs: [[url:
            'file:///Users/jmckenzie/projects/santa']]]
        sh "sh reindeer.sh ${greeting} ${reindeer}"
      }
    }
  }
  catch (Throwable err) {
    throw err
  }
}
```

We have our pipeline code –
One instance –
Shared across all of Jenkins,
Versioned, In source control

@jeffreymckenzie

```
- Jenkinsfile
#!groovy
@Library('jenkins-lib') _

options = [
    greeting: GREETING,
    reindeer: REINDEER
]
reindeer(options)
```

We have our Jenkinsfile,
Again, once instance of it,
Versioned, and in source control



And finally, our one DSL job and script
So to scale this out further
You only really need to add an entry in that array.

So all of that in about the same amount
Of code that you probably throw away each day.

Insight. Digital Innovation

Automating Your Automation

The Care and Feeding of Jenkins

Jeff McKenzie • jeff.mckenzie@insight.com • @jeffreymckenzie

??????