Changes Documentation

Release 0.1.0

Dropbox, Inc.

October 14, 2016

Contents

1 Users Guide **3**
- 1.1 Setup Guide . . . 3
- 1.2 Using Changes . . . 5
- 1.3 Jenkins Integration . . . 8
- 1.4 Phabricator Integration . . . 12

2 Developers **13**
- 2.1 Contributing to Changes . . . 13
- 2.2 Snapshotting . . . 18
- 2.3 Data Model . . . 19

3 Resources **25**

Changes is a build coordinator and reporting solution written in Python.

The project is primarily built on top of Jenkins, but efforts are underway to replace the underlying dependency. The current work-in-progress tooling exists under several additional repositories:

- https://github.com/dropbox/changes-client
- https://github.com/dropbox/changes-mesos-framework

CHAPTER 1

Users Guide

1.1 Setup Guide

1.1.1 Getting the Source Code

Use the git, Luke!

```
$ git clone https://github.com/dropbox/changes.git
```

1.1.2 Installing Dependencies

We're going to assume you're running OS X, otherwise you're on your own.

```
$ brew install node libev libxml2 libxslt python
```

Install Postgres (ensure you follow whatever instructions are given post-install):

```
$ brew install postgresql
```

Install Redis (ensure you follow whatever instructions are given post-install):

```
$ brew install redis
```

Next up, we need Bower for JavaScript dependencies:

```
$ npm install -g bower
```

And finally let's make sure we have virtualenv for our Python environment:

```
$ pip install --upgrade virtualenv
```

1.1.3 Configure the Environment

Create the database in Postgres:

```
$ createdb -E utf-8 changes
```

Setup the default configuration:

```
# ~/.changes/changes.conf.py
WEB_BASE_URI = 'http://localhost:5000'
INTERNAL_BASE_URI = 'http://localhost:5000'
SERVER_NAME = 'localhost:5000'

REPO_ROOT = '/tmp'

# You can obtain these values via the Google Developers Console:
# https://console.developers.google.com/
# Example 'Authorized JavaScript Origins': http://localhost:5000
# Example 'Authorized Redirect URIs': http://localhost:5000/auth/complete/
GOOGLE_CLIENT_ID = None
GOOGLE_CLIENT_SECRET = None
```

Create a Python environment:

```
# set cwd to repo root
$ cd /path/to/changes

# create a base environment
$ virtualenv env

# "active" the environment, so python becomes localized
$ source env/bin/activate
```

Bootstrap your environment:

```
# fix for Xcode 5.1
$ export ARCHFLAGS=-Wno-error=unused-command-line-argument-hard-error-in-future

# install basic dependencies (npm, bower, python)
$ make develop

# if not launched automatically, manually start the needed servers
$ postgres -D /usr/local/var/postgres &
$ redis-server /usr/local/etc/redis.conf &

# perform any data migrations
$ make upgrade
```

Take a glance at the Makefile for more details on what commands are available, and what actually gets executed.

1.1.4 Installing Services

You're going to need to run several services in the background. Specifically, you'll need both the webserver and the workers running. To do this we recommend using supervisord.

Below is a sample configuration for both the web and worker processes:

```
    [program:changes-web]
    command=/srv/changes/env/bin/uwsgi --http 127.0.0.1:50%(process_num)02d --processes 1 --threads
    user=ubuntu
    environment=CHANGES_CONF="/srv/changes/config.py",PATH="/srv/changes/env/bin:/usr/local/sbin:/us
    process_name=%(program_name)s_%(process_num)02d
    numprocs=4
    autorestart=true
    killasgroup=true
    stopasgroup=true
```

```
directory=/srv/changes
redirect_stderr=true
stdout_logfile=/tmp/%(program_name)s_%(process_num)02d.log

[program:changes-worker]
command=/srv/changes/env/bin/celery -A changes.app:celery worker -c 96 --without-mingle
user=ubuntu
environment=CHANGES_CONF="/srv/changes/config.py",PATH="/srv/changes/env/bin:/usr/local/sbin:/us
directory=/srv/changes
autorestart=true
killasgroup=true
stopasgroup=true
redirect_stderr=true
stdout_logfile=/tmp/%(program_name)s_%(process_num)02d.log
```

For more details you'll want to refer to the supervisord documentation.

1.2 Using Changes

At a high level, Changes is designed to be a view into the lifetime of a changeset. This is mostly apparent when you're viewing the results of automated tests.

A few concepts you should understand about Changes:

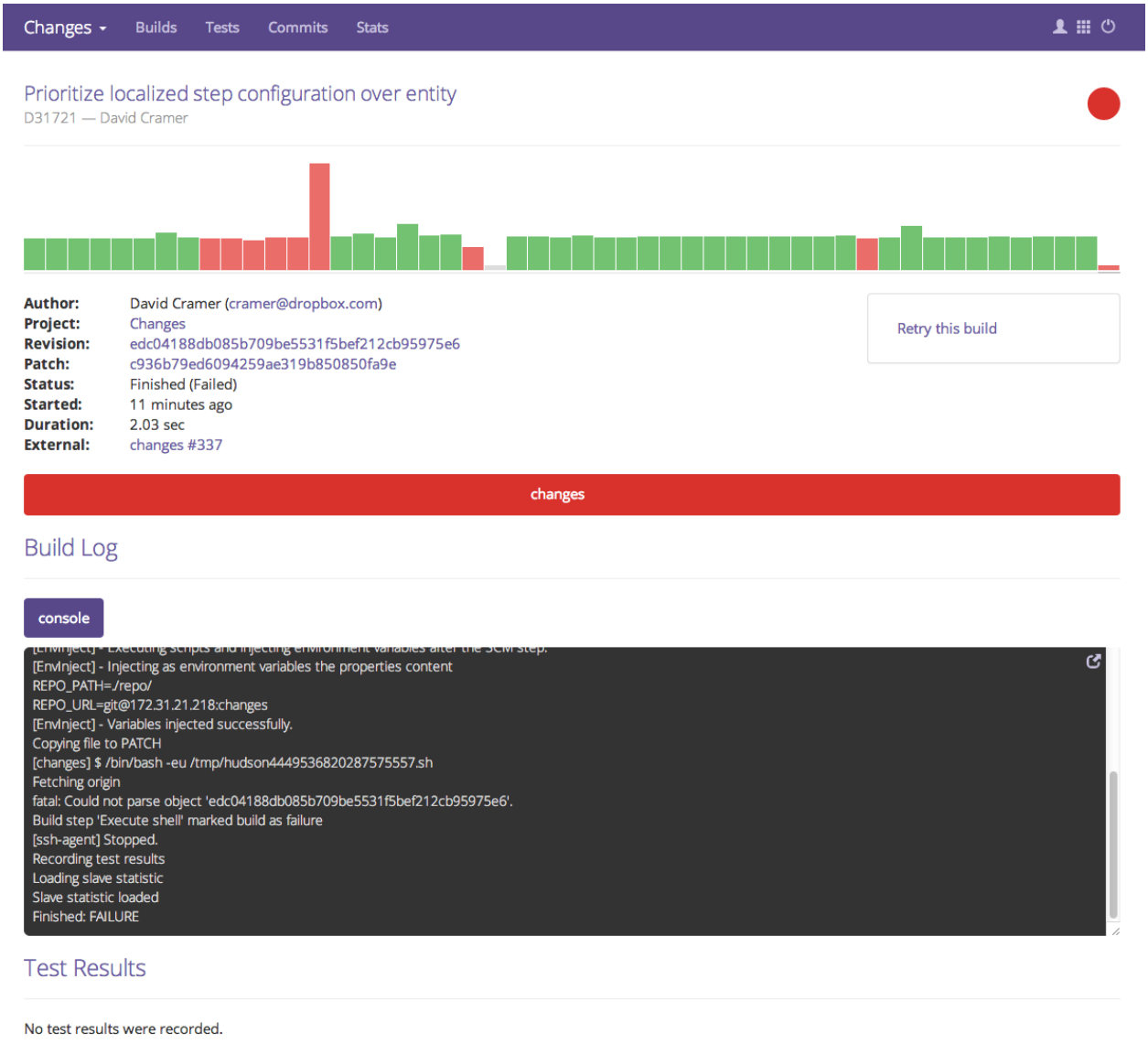
- A "Change" is a discrete object that represents a code changeset. This generally starts out as a patch, and works its way into a commit.
- Builds are bundled into families, and each build target has it's own build. That is if your project needs to build on two platforms, there are two separate builds bundled into a single family grouping.

1.2.1 Understanding Build Results

To get the most out of Changes, the first thing you'll want to dive into are build results. These are presented in a variety of ways, but the most common forms of interaction will be via **email** and the **build details report**.

Data available depends on what the build was able to report. In some cases you may only be able to rely on the log output. In general, if things are working as intended, you'll be able to simply drill into an individual test result to see what's going on.

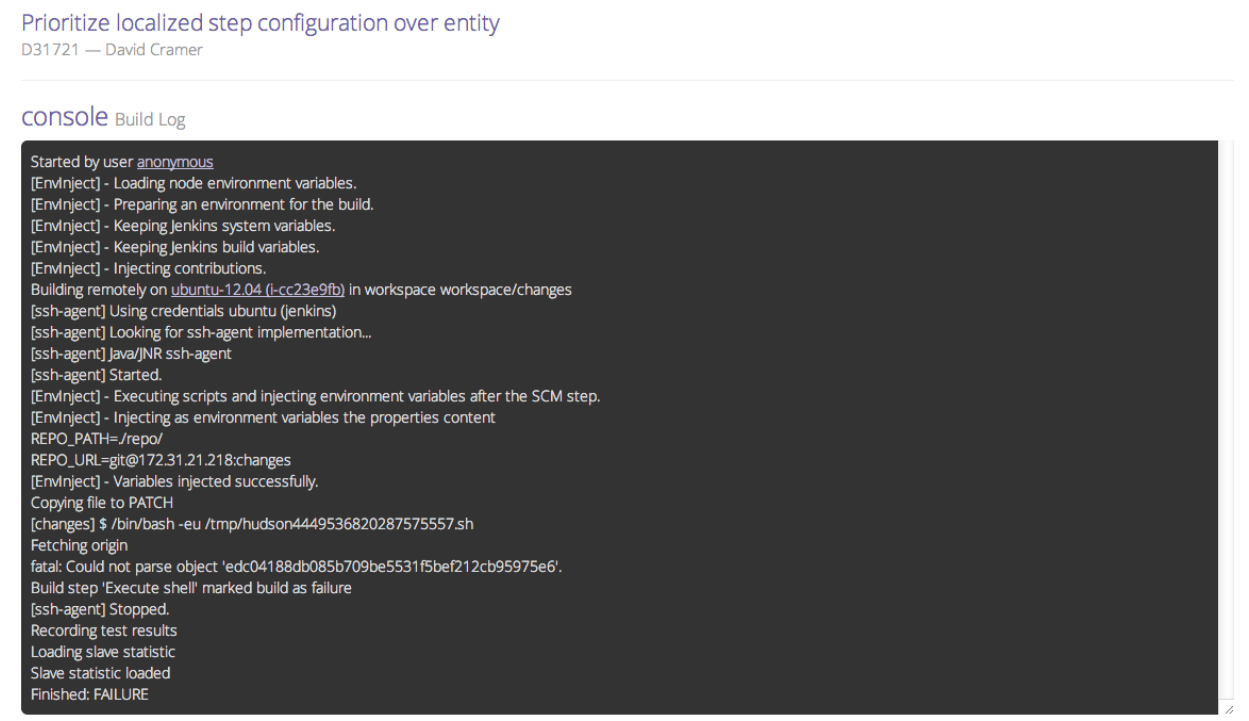
Let's start by taking a look at the minimal build report:



A few key things here:

- The phase that was executed (changes). There's only one in our example, but different systems may provide more details.
- The log streams available. In our example only a single console log was recorded.
- Test results. In this case we hit a critical failure, so no test results were reported.

So looking into how we'd understand this result, we're probably going to want to expand the build log. You can do this via the icon in the top right corner of the log stream. Once you're in here, it should be much easier to visually spot the failure, in our case it looks like we failed on checking out the revision:



Now at the same time we also received an email as we were the author of this commit. The email contains a lot of similar information, but in a more limited fashion:

Prioritize localized step configuration over entity D31721

Build: 1a763cfe3adf4672ba8849f48113d231
Author: David Cramer

Build Log console

```
Started by user <a href='/user/null' class='model-link'>anonymous</a>
[EnvInject] - Loading node environment variables.
[EnvInject] - Preparing an environment for the build.
[EnvInject] - Keeping Jenkins system variables.
[EnvInject] - Keeping Jenkins build variables.
[EnvInject] - Injecting contributions.
Building remotely on <a href='/computer/ubuntu-12.04 (i-cc23e9fb)' class='model-link'>ubuntu-12.04
[ssh-agent] Using credentials ubuntu (jenkins)
[ssh-agent] Looking for ssh-agent implementation...
[ssh-agent]   Java/JNR ssh-agent
[ssh-agent] Started.
[EnvInject] - Executing scripts and injecting environment variables after the SCM step.
[EnvInject] - Injecting as environment variables the properties content
REPO_PATH=./repo/
REPO_URL=git@172.31.21.218:changes

[EnvInject] - Variables injected successfully.
Copying file to PATCH
[changes] $ /bin/bash -eu /tmp/hudson4449536820287575557.sh
Fetching origin
fatal: Could not parse object 'edc04188db085b709be5531f5bef212cb95975e6'.
Build step 'Execute shell' marked build as failure
[ssh-agent] Stopped.
Recording test results
Loading slave statistic
Slave statistic loaded
Finished: FAILURE
```

View full log

As you can see in this case, all we've got available is the build log, so it's up to us to scan through and identify the issue.

1.3 Jenkins Integration

Changes integrates extremely with Jenkins as a build manager, however it will require you to have a very specialized job for running a build.

1.3.1 Creating the Job

This changes rapidly and documentation is not maintained for the internals of the generic job.

```
<?xml version='1.0' encoding='UTF-8'?>
<project>
  <actions/>
  <description></description>
  <logRotator class="hudson.tasks.LogRotator">
    <daysToKeep>7</daysToKeep>
    <numToKeep>1000</numToKeep>
    <artifactDaysToKeep>-1</artifactDaysToKeep>
    <artifactNumToKeep>100</artifactNumToKeep>
  </logRotator>
  <keepDependencies>false</keepDependencies>
  <properties>
    <hudson.model.ParametersDefinitionProperty>
      <parameterDefinitions>
        <hudson.model.StringParameterDefinition>
          <name>REVISION</name>
          <description>A commit, branch name, or something else recognizable as a tree.
</description>
          <defaultValue></defaultValue>
        </hudson.model.StringParameterDefinition>
        <hudson.model.StringParameterDefinition>
          <name>PATCH_URL</name>
          <description></description>
          <defaultValue></defaultValue>
        </hudson.model.StringParameterDefinition>
        <hudson.model.StringParameterDefinition>
          <name>REPO_URL</name>
          <description></description>
          <defaultValue></defaultValue>
        </hudson.model.StringParameterDefinition>
        <hudson.model.ChoiceParameterDefinition>
          <name>REPO_VCS</name>
          <description></description>
          <choices class="java.util.Arrays$ArrayList">
            <a class="string-array">
              <string>git</string>
              <string>hg</string>
            </a>
          </choices>
        </hudson.model.ChoiceParameterDefinition>
        <hudson.model.StringParameterDefinition>
          <name>SCRIPT</name>
          <description></description>
          <defaultValue></defaultValue>
        </hudson.model.StringParameterDefinition>
        <hudson.model.StringParameterDefinition>
          <name>RESET_SCRIPT</name>
          <description>Cleanup tasks to run after the generic job completes. Intended for use by rese
          </description>
          <defaultValue></defaultValue>
        </hudson.model.StringParameterDefinition>
        <hudson.model.StringParameterDefinition>
          <name>CHANGES_PID</name>
          <description>Changes Project Slug</description>
          <defaultValue></defaultValue>
```

```
        </hudson.model.StringParameterDefinition>
        <hudson.model.StringParameterDefinition>
          <name>CHANGES_BID</name>
          <description>Changes Job ID (DO NOT ENTER MANUALLY)</description>
          <defaultValue></defaultValue>
        </hudson.model.StringParameterDefinition>
        <org.jvnet.jenkins.plugins.nodelabelparameter.LabelParameterDefinition plugin="nodelabelparam
          <name>CLUSTER</name>
          <description></description>
          <defaultValue></defaultValue>
          <allNodesMatchingLabel>false</allNodesMatchingLabel>
          <triggerIfResult>allCases</triggerIfResult>
          <nodeEligibility class="org.jvnet.jenkins.plugins.nodelabelparameter.node.AllNodeEligibilit
        </org.jvnet.jenkins.plugins.nodelabelparameter.LabelParameterDefinition>
        <hudson.model.StringParameterDefinition>
          <name>WORK_PATH</name>
          <description>Working directory to run SCRIPT from. Relative to REPO_ROOT.</description>
          <defaultValue></defaultValue>
        </hudson.model.StringParameterDefinition>
        <hudson.model.StringParameterDefinition>
          <name>C_WORKSPACE</name>
          <description>Custom workspace directory.</description>
          <defaultValue></defaultValue>
        </hudson.model.StringParameterDefinition>
      </parameterDefinitions>
    </hudson.model.ParametersDefinitionProperty>
    <com.sonyericsson.rebuild.RebuildSettings plugin="rebuild@1.20">
      <autoRebuild>true</autoRebuild>
    </com.sonyericsson.rebuild.RebuildSettings>
  </properties>
  <scm class="hudson.scm.NullSCM"/>
  <canRoam>true</canRoam>
  <disabled>false</disabled>
  <blockBuildWhenDownstreamBuilding>false</blockBuildWhenDownstreamBuilding>
  <blockBuildWhenUpstreamBuilding>false</blockBuildWhenUpstreamBuilding>
  <authToken/>
  <triggers/>
  <concurrentBuild>true</concurrentBuild>
  <builders>
    <hudson.tasks.Shell>
      <command>#!/bin/bash -ex

/var/lib/jenkins/build-steps/generic-build
</command>
    </hudson.tasks.Shell>
  </builders>
  <publishers>
    <hudson.tasks.ArtifactArchiver>
      <artifacts>**/junit.xml,**/coverage.xml,**/tests.json,**/jobs.json</artifacts>
      <latestOnly>false</latestOnly>
      <allowEmptyArchive>true</allowEmptyArchive>
    </hudson.tasks.ArtifactArchiver>
    <hudson.tasks.BuildTrigger>
      <childProjects>reset-generic</childProjects>
      <threshold>
        <name>FAILURE</name>
        <ordinal>2</ordinal>
        <color>RED</color>
```

```
        <completeBuild>true</completeBuild>
      </threshold>
    </hudson.tasks.BuildTrigger>
  </publishers>
</project>
```

Example scripts based on git are included for reference. Note that REPO_PATH is a global variable that is assumed to exist. The reset-generic job here is an optional, sample downstream job that can be run to execute cleanup tasks passed through the RESET_SCRIPT parameter. Running cleanup tasks outside the generic job has the advantage of not delaying build results from the generic job being posted back to Changes.

Master Build Step

```
#!/bin/bash -ex

echo `whoami`@$HOSTNAME
uname -a

# nothing works without ssh keys, so let's straight up error
# out of theres no keys/agent present
ssh-agent -s
ssh-add -l

REPO_PATH=$WORKSPACE/$CHANGES_PID

if [ -z $REVISION ]; then
  if [ "$REPO_VCS" = "hg" ]; then
    REVISION=default
  else
    REVISION=master
  fi
fi

if [ "$REPO_VCS" = "git" ]; then
  git-clone $REPO_PATH $REPO_URL $REVISION
  git-patch $REPO_PATH $PATCH_URL
else
  hg-clone $REPO_PATH $REPO_URL $REVISION
  hg-patch $REPO_PATH $PATCH_URL
fi

# clean up any artifacts which might be present
for artifact_name in "junit.xml coverage.xml jobs.json tests.json"; do
        find . -name $artifact_name -delete
done

SCRIPT_PATH=/tmp/$(mktemp build-step.XXXXXXXXXX)
echo "$SCRIPT" | tee $SCRIPT_PATH
chmod +x $SCRIPT_PATH

pushd $REPO_PATH

if [ ! -z $WORK_PATH ]; then
    pushd $WORK_PATH
fi

$SCRIPT_PATH
```

Fetching the Revision

```
#!/bin/bash -eux

if [ ! -d $REPO_PATH/.git ]; then
        git clone $REPO_URL $REPO_PATH
        pushd $REPO_PATH
else
        pushd $REPO_PATH && git fetch --all
        git remote prune origin
fi

git clean -fdx

if ! git reset --hard $REVISION ; then
        git reset --hard origin/master
        echo "Failed to update to $REVISION, falling back to master"
fi
```

Applying the Patch

```
#!/bin/bash -eux

WORKSPACE_DIR=$(pwd)

pushd $REPO_PATH
if [ ! -z "${PATCH_URL:-}" ]; then
        curl -o ${WORKSPACE_DIR}/PATCH $PATCH_URL
        git apply ${WORKSPACE_DIR}/PATCH
fi
```

Running Tests

This step is arbitrary based on your platform. In Python this might be something like:

```
py.test --junit=junit.xml
```

1.4 Phabricator Integration

While Changes provides no direct integration with Phabricator, a build step is available as an external library:

https://github.com/dropbox/phabricator-changes

The build step allows you to wire up your Phabricator install to submit commits and diffs to Changes as builds.

CHAPTER 2

Developers

2.1 Contributing to Changes

To get started, either get a job at Dropbox, or sign our CLA:

https://opensource.dropbox.com/cla/

2.1.1 Setup

Start by configuring your environment per the setup guide. You can safely skip the various production requirements (such as setting up services).

Webserver

Run the webserver:

```
$ bin/web
```

Note: The server doesn't automatically reload when you make changes to the Python code.

Background Workers

While it's likely you won't need to actually run the workers, they're managed via Celery.

```
# Start a generic worker process
# the -B flag indicates to also start "celerybeat" which
# is utilized for periodic tasks.
$ bin/worker -B
```

Note: In development you can set `CELERY_ALWAYS_EAGER=True` to run the queue tasks synchronously in-process. Generally we prefer to test through automated integration tests, but this is useful if you want to QA and don't want to run several processes.

2.1.2 Directory Layout

While there are a significant and growing number of paths, this is an attempt to outline some of the more common and important code paths.

```
# command line scripts
-- bin

# python code
-- changes

# the core of url registration and app configuration
|   -- config.py

# api controllers and serializers
|   -- api

# various integration code (primarily legacy for communicating with Jenkins)
|   -- backends

# implementations of the various buildsteps (modern build handlers)
|   -- buildsteps

# database utilities
|   -- db

# implementations of build factory expanders
|   -- expanders

# tasks executed asynchronously via Celery workers
|   -- jobs

# our sqlalchemy model definitions
|   -- models

# integration code for mercurial/git
|   -- vcs

# python test bootstrap code
-- conftest.py

# docs, like what you're reading right now
-- docs

# database migrations (via Alembic)
-- migrations

# client-side templates
-- partials

# static media (such as the frontend code, as well as vendored code within)
-- static
|   -- css
|   -- js
|   -- vendor

# server-side templates
-- templates
```

```
# all tests (only python currently)
-- tests
```

2.1.3 Understanding the Frontend

Everything is bundled into a "state". A state is a combination of a router and a controller, and it contains nearly all of the logic for rendering an individual page.

States are registered into routes.js (they get required and then registered to a unique name).

As an example, let's take a look at planList.js, a fairly simple state:

```
// static/js/states/planList.js
define(['app'], function(app) {
  'use strict';

  return {
    // parent is used for template/scope inheritance
    parent: 'layout',

    // the url **relative** to the parent
    // in our case, layout is the parent which has no base url
    url: '/plans/',

    // all templates exist in partials/
    templateUrl: 'partials/plan-list.html',

    // $scope, planList, and Collection are all dependencies, implicitly
    // parsed by angular and included in the function's scope
    controller: function($scope, planList, Collection) {
      // binding to $scope adds it to the template context
      $scope.plans = new Collection(planList);
    },

    // resolvers get executed **before** the controller is run and
    // are ideal for loading initial data
    resolve: {
      planList: function($http) {
        // this **must** return a future
        return $http.get('/api/0/plans/').then(function(response){
            return response.data;
        });
      }
    }
  };
});
```

Then within routes.js, we register this under the 'plan_list' namespace:

```
// static/js/routes.js
define([
  'app',
  'states/layout',
  // ...
  'states/planList'
], function(
  // the order of dependencies must match above
  app,
```

```
  LayoutState,
  // ...
  PlanListState
) {
  // this has been simplified for illustration purposes
  app.config(function($stateProvider) {
  $stateProvider
    .state('layout', LayoutState)
    // ...
    .state('plan_list', PlanListState);
});
```

Let's take a look at the template, plan-list.html:

```
<!-- partials/plan-list.html -->
<section ui-view>
    <div id="overview">
        <div class="page-header">
            <h2>Build Plans</h2>
        </div>

        <table class="table table-striped">
            <thead>
                <tr>
                    <th>Plan</th>
                    <th style="width:150px;text-align:center">Created</th>
                    <th style="width:150px;text-align:center">Modified</th>
                </tr>
            </thead>
            <tbody>
                <tr ng-repeat="plan in plans">
                    <td><a ui-sref="plan_details({plan_id: plan.id})">{{plan.name}}</a></td>
                    <td style="text-align:center" time-since="plan.dateCreated"></td>
                    <td style="text-align:center" time-since="plan.dateModified"></td>
                </tr>
            </tbody>
        </table>
    </div>
</section>
```

There's a few key things to understand in this simple example:

```
<section ui-view>
```

The ui-view attribute here is what Angular calls a directive. In this case, it actually maps to the library we use (ui-router) and says "content within this can be replaced by the child template". That's not precisely the meaning, but for our examples it's close enough.

Jumping down to actual rendering:

```
<tr ng-repeat="plan in plans">
```

This is another built-in directive, and it says "expand 'plans', and assign the item at the current index to 'plan'".

We can then reference it:

```
<td><a ui-sref="plan_details({plan_id: plan.id})">{{plan.name}}</a></td>
```

Two things are happening here:

- We're specifying ui-sref, which is saying "find the named url with these parameters". Parameters are always inherited, so you only need to pass in the changed values.
  - In our specific example, we're referring to the `plan_details` state, which might be a child page of `plan_list`. This is the same name you would define in the `.state()` registration.
  - We also need to pass the `plan_id` parameter, which is used by the state's url matcher, and then made available via `$stateParams` within it's controller.
- Render the `name` attribute of this plan.

There's also a couple uses of our timeSince.js directive:

```
<td style="text-align:center" time-since="plan.dateCreated"></td>
```

In most uses of directives, you'll notice that we don't surround the value with `{{ }}`. This is because the directive itself is choosing to evaluate the value as part of the scope.

2.1.4 Understanding the Backend

The backend is a fairly straightforward Flask app. It has two primary models: task execution and consumer API.

We're not going to explain the workers as they contain a very large amount of coordination logic, but instead let's focus on the API.

To start with, the entry point for URLs currently lives in `config.py`, under `configure_api_routes`. You'll see that each API controller lives in a separate module space and is registered into the routing here.

Let's take a look at the API controller for our `plan_list` state, contained in plan_index.py:

```
# changes/api/plan_index.py
from __future__ import absolute_import, division, unicode_literals

from changes.api.base import APIView
from changes.models import Plan


class PlanIndexAPIView(APIView):
    def get(self):
        results = Plan.query.order_by(Plan.label.asc())[:10]

        # while respond() can serialize for you, we use this for illustration
        # purposes
        data = self.serialize(results)

        return self.respond(data, serialize=False)
```

There's no real surprises here if you've ever written Python. We're using SQLAlchemy to query the `Plan` table, and we're returning a simple result of ten plans.

There are two things happening here:

- We're serializing the list of Plans using the default registered serializer (dig into the serializer to see what this does.)
- `respond()` is then going to return an HTTP response object, with a 200 status code any required headers, as well as eventually encode our Python object into JSON.

And of course, we absolutely require integration tests for every endpoint, which live in test_plan_index.py:

```
from changes.testutils import APITestCase


class PlanIndexTest(APITestCase):
    path = '/api/0/plans/'

    def test_simple(self):
        plan1 = self.plan
        plan2 = self.create_plan(label='Bar')

        resp = self.client.get(self.path)
        assert resp.status_code == 200
        data = self.unserialize(resp)
        assert len(data) == 2
        assert data[0]['id'] == plan2.id.hex
        assert data[1]['id'] == plan1.id.hex
```

A `client` attribute exists on the test instance, as well as a number of helpers in changes.testutils.fixtures for creating mock data. This is a real database transaction so you can do just about everything, and we'll safely ensure things are cleaned up and fast.

2.1.5 Loading in Mock Data

If you're changing the frontend, it's likely you're going to want some data to work with. We've provided a helper script which will create some sample data, as well as stream in continuous updates. It's not quite the same as production, but it should be enough to work with:

```
$ python stream_data.py
```

2.2 Snapshotting

One of the core concepts of Changes is snapshotting. At a high level, the API only provides an abstraction for snapshots and relies on individual adapters to actually determine what they mean.

2.2.1 Architecture

Snapshots consists of two discrete data models: Snapshot (snapshot) and SnapshotImage (snapshot_image).

A Snapshot object itself acts on a single Source (i.e. a commit) and generates an image for each build plan associated with a given project. This means that a SnapshotImage is keyed on (snapshot_id, plan_id).

For example, we might have a build that has two build plans: precise and lucid (two distros of ubuntu). You'd generally see a single build created, and within it two jobs: one job for precise, and one job for lucid. When a snapshot is generated, it will behave similarly, and it will also create two SnapshotImage objects: one for precise, and one for lucid.

2.2.2 The Build Process

When a build is created for a snapshot it will be registered in the system with a special Cause attribute (linked to Cause.snapshot). A snapshot will be put into a pending state initially, and when all images are successfully built (again, one per plan) the snapshot will become 'active' and can then be set as the default snapshot for a given project.

If any of the jobs failed the entire snapshot is considered invalid and cannot be used.

If a new build plan is added when an existing snapshot is activated, it will simply ignore the missing image for the new build plan, which would suggest to the underlying system that they use whatever the default is.

2.3 Data Model

class changes.models.artifact.**Artifact**(***kwargs*)
The artifact produced by one job/step, produced on a single machine. Sometimes this is a JSON dict referencing a file in S3, sometimes it is Null, sometimes it is an empty dict. It is basically any file left behind after a run for changes to pick up

class changes.models.author.**Author**(***kwargs*)
A list of every person who has written a revision parsed by changes. Keyed by email. Automatically updated when new authors are seen by changes in diffs etc.

class changes.models.build.**Build**(***kwargs*)
Represents the work we do (e.g. running tests) for one diff or commit (an entry in the source table) in one particular project

Each Build contains many Jobs (usually linked to a JobPlan).

class changes.models.buildseen.**BuildSeen**(***kwargs*)
Keeps track of when users have viewed builds in the ui. Not sure we expose this to users in the ui right now.

class changes.models.command.**Command**(***kwargs*)
The information of the script run on one node within a jobstep: the contents of the script are included, and later the command can be updated with status/return code.

changes-client has no real magic beyond running commands, so the list of commands it ran basically tells you everything that happened.

Looks like only mesos/lxc builds (DefaultBuildStep)

class changes.models.comment.**Comment**(***kwargs*)
Comments on test runs in changes. You can go into the GUI and leave messages, and this table keeps track of those. There is a job_id but it is always null, despite the UI showing you the comment only on the job page.

Due to this, the UI will show an identical set of comments on every job page of a build.

class changes.models.event.**Event**(***kwargs*)
Indicates that something (specified by *type* and *data*) happened to some entity (specified by *item_id*). This allows us to record that we've performed some action with an external side-effect so that we can be sure we do it no more than once. It is also useful for displaying to users which actions have been performed when, and whether they were successful.

class changes.models.failurereason.**FailureReason**(***kwargs*)
Always associated with a single jobstep. failurereason is not required to fail a build. But if a jobstep fails, it can record why here. reason column can be: [test_failures, missing_test, missing_artifact, timeout, malformed_artifact, duplicate_test_name]

class changes.models.filecoverage.**FileCoverage**(***kwargs*)
Unique to file/job/project. Contains a data-blob-string, where each character is either

- U Unconvered
- C Covered
- N No Info

filled in when file coverage artifacts are collected (updated with additional lines for each new artifact in a job)

class changes.models.itemsequence.**ItemSequence**(***kwargs*)
Used to hold counters for autoincrement-style sequence number generation. In each row, value is the last sequence number returned for the corresponding parent.

The table is used via the next_item_value database function and not used in the python codebase.

class changes.models.itemstat.**ItemStat**(***kwargs*)
Also a key/value table, tailored towards statistics generated by tests and code coverage. Examples: test_rerun_count, test_duration, lines_covered

class changes.models.job.**Job**(***kwargs*)
An instantiation of a plan for a particular build. We run the code specified by the appropriate plan. That code creates and farms out a bunch of jobsteps to do the actual work.

class changes.models.jobplan.**JobPlan**(***kwargs*)
A snapshot of a plan and its constituent steps, taken at job creation time. This exists so that running jobs are not impacted by configuration changes. Note that this table combines the data from the plan and step tables.

class changes.models.jobstep.**JobStep**(***kwargs*)
The most granular unit of work; run on a particular node, has a status and a result.

class changes.models.jobphase.**JobPhase**(***kwargs*)
A JobPhase is a grouping of one or more JobSteps performing the same basic task. The phases of a Job are intended to be executed sequentially, though that isn't necessarily enforced.

One example of phase usage: a Job may have a test collection phase and a test execution phase, with a single JobStep collecting tests in the first phase and an arbitrary number of JobSteps executing shards of the collected tests in the second phase. By using two phases, the types of JobSteps can be tracked and managed independently.

Though JobPhases are typically created to group newly created JobSteps, they can also be constructed retroactively once a JobStep has finished based on phased artifacts. This is convenient but a little confusing, and perhaps should be handled by another mechanism.

class changes.models.latest_green_build.**LatestGreenBuild**(***kwargs*)
Represents the latest green build for a given branch of a given project

A project with multiple latest_green_builds is because it has multiple branches

class changes.models.log.**LogSource**(***kwargs*)
We log the console output for each jobstep. logsource is an entity table for these "logfiles". logchunk contains the actual text.

If we're using artifact store to store/host the log file, in_artifact_store will be set to true. No logchunk entries will be associated with such logsources.

class changes.models.log.**LogChunk**(***kwargs*)
Chunks of text. Each row in logchunk is associated with a particular logsource entry, and has an offset and blob of text. By grabbing all logchunks for a given logsource id, you can combine them to get the full log.

class changes.models.node.**Cluster**(***kwargs*)
A group of nodes. We refer to clusters in the step configurations (where should we run our tests?) Clusters are automatically added when we see them from jenkins results.

Apparently, clusters are only used in jenkins (not lxc, although nodes are used for both.) A cluster does not correspond to one master

class changes.models.node.**ClusterNode**(*cluster=None*, *node=None*, ***kwargs*)
Which cluster does each node belong to? This is populated at the same time as cluster.

class changes.models.node.**Node**(**args*, ***kwargs*)
A machine that runs jobsteps.

This is populated by observing the machines picked by the jenkins masters (which themselves are configured by BuildStep params in the changes UI) when they're asked to run task, and is not configured manually. Node machines have tags (not stored in the changes db)

class changes.models.patch.**Patch**(***kwargs*)
A patch that can be applied to a revision. Refers to a parent revision on which the patch is based, and contains a diff text field with the contents of the patch (in unified diff form? 2x check.)

Used by builds from phabricator diffs: see source for more details.

class changes.models.phabricatordiff.**PhabricatorDiff**(***kwargs*)
Whenever phabricator sends us a diff to do a build against (see source/patch for more info), we write an entry to this table with the details. revision_id and diff_id refer to the phabricator versions of this terminology: revision_id is the number in D145201 and diff_id represents a particular diff within that differential revision (the id in the revision update history table.)

This is 80% convenient logging. It also does light deduplication: we make sure to never kick off more than one build for a particular revision_id/diff_id from the api called by phabricator. Phabricator can occasionally fire a herald rule more than once, so its nice to have this.

class changes.models.plan.**Plan**(***kwargs*)
What work should we do for our new revision? A project may have multiple plans, e.g. whenever a diff comes in, test it on both mac and windows (each being its own plan.) In theory, a plan consists of a sequence of steps; in practice, a plan is just a wrapper around a single step.

class changes.models.project.**Project**(***kwargs*)
The way we organize changes. Each project is linked to one repository, and usually kicks off builds for it when new revisions come it (or just for some revisions based on filters.) Projects use build plans (see plan) to describe the work to be done for a build.

class changes.models.project.**ProjectOption**(***kwargs*)
Key/value table storing configuration information for projects. Here is an incomplete list of possible keys:

- build.branch-names
- build.commit-trigger
- build.expect-tests
- build.file-whitelist
- build.test-duration-warning
- green-build.notify
- green-build.project
- history.test-retention-days
- mail.notify
- mail.notify-addresses
- mail.notify-addresses-revisions
- mail.notify-author
- phabricator.diff-trigger
- phabricator.notify
- phabricator.coverage
- project.notes
- project.owners

- snapshot.current
- ui.show-coverage
- ui.show-tests

class changes.models.repository.**Repository**(***kwargs*)
Represents a VCS repository that Changes will watch for new commits.

class changes.models.revision.**Revision**(***kwargs*)
Represents a commit in a repository, including some metadata. Author and committer are stored as references to the author table. Ideally there will be one revision row for every commit in every repository tracked by changes, though this is not always true, and some code tries to degrade gracefully when this happens.

Revisions are keyed by repository, sha. They do not have unique UUIDs

class changes.models.snapshot.**Snapshot**(***kwargs*)
A snapshot is a set of LXC container images (up to one for each plan in a project).

Each project can have an arbitrary number of snapshots, but only up to one "current snapshot" is actually used by builds (stored as ProjectOption) at any time.

Snapshots are used in the Mesos and Jenkins-LXC environments. Snapshots are currently only used with changes-client.

When running a build, the images of the current snapshot are used for individual jobs that are part of a build. A snapshot image can be shared between multiple plans by setting snapshot_plan_id of a Plan. By default, there is a separate image for each plan of a build.

The status of a snapshot indicates whether it *can* be used for builds; it doesn't indicate whether the snapshot is actually used for builds right now. A snapshot is active if and only if all the corresponding snapshot images are active.

A snapshot is generated by a slightly special snapshot build that uploads a snapshot at the end of the build.

class changes.models.source.**Source**(***kwargs*)
This is the object that actually represents the code we run builds against.

Essentially its a revision, with a UUID, and a possible patch_id. Rows with null patch_ids are just revisions, and rows with patch_ids apply the linked patch on top of the revision and run builds against the resulting code.

Why the indirection? This is how we handle phabricator diffs: when we want to create a build for a new diff, we add a row here with the diff's parent revision sha (NOT the sha of the commit phabricator is trying to land, since that will change every time we update the diff) and a row to the patch table that contains the contents of the diff.

Side note: Whenever we create a source row from a phabricator diff, we log json text to the data field with information like the diff id.

class changes.models.step.**Step**(***kwargs*)
A specific description of how to do some work for a build.

In theory, a plan can have multiple steps. In practice, every plan has only one step and plan is just a thin wrapper around step. Steps are not freeform, rather, each step is just configuration data for specific step implementations that are hard-coded in python.

class changes.models.task.**Task**(***kwargs*)
When we enqueue a task, we also write a db row to keep track of the task's metadata (e.g. number of times retried.) There is a slightly icky custom data column that each task type uses in its own way. This db represents serialized version of tracked_task you see in the changes python codebase.

Tasks can have parent tasks. Parent tasks have the option of waiting for their children to complete (in practice, that always happens.)

Example: sync_job with sync_jobstep children

Tasks can throw a NotFinished exception, which will just mean that we try running it again after some interval (but this has nothing to do with retrying tasks that error!) Examples: Tasks with children will check to see if their children are finished; the sync_jobstep task will query jenkins to see if its finished.

Tasks can fire signals, e.g. build xxx has finished. There's a table that maps signal types to tasks that should be created. Signals/listeners are not tracked as children of other tasks.

See *Tasks* for more details on what the task_name can refer to.

class changes.models.test.**TestCase**(***kwargs*)
A single run of a single test, together with any captured output, retry-count and its return value.

Every test that gets run ever has a row in this table.

At the time this was written, it seems to have 400-500M rows

(how is this still surviving?)

NOTE: DO NOT MODIFY THIS TABLE! Running migration on this table has caused unavailability in the past. If you need to add a new column, consider doing that on a new table and linking it back to tests via the ID.

class changes.models.testartifact.**TestArtifact**(***kwargs*)
Represents any artifacts generated by a single run of a single test. used e.g. in server-selenium to store screenshots and large log dumps for later debugging.

class changes.models.user.**User**(***kwargs*)
A table of the people who use changes.

2.3.1 Tasks

changes.jobs.create_job.**create_job**()
Kicks off a newly created job within a build; enqueued for each job within a new build.

changes.jobs.import_repo.**import_repo**()

changes.jobs.signals.**fire_signal**()
Tasks fire signals by spawning fire_signal tasks; they grab every associated listener and spawn run_event_listener tasks for each

changes.jobs.signals.**run_event_listener**()
Actually run the listener

See fire_signal, which doesn't actually run it

changes.jobs.sync_artifact.**sync_artifact**()
Downloads an artifact from jenkins.

changes.jobs.sync_job.**sync_job**()
Updates jobphase and job statuses based on the status of the constituent jobsteps.

changes.jobs.sync_job_step.**sync_job_step**()
Polls a build for updates. May have sync_artifact children.

changes.jobs.sync_repo.**sync_repo**()

CHAPTER 3

Resources

- Bug Tracker
- Code
- CLA

A

Artifact (class in changes.models.artifact), 19
Author (class in changes.models.author), 19

B

Build (class in changes.models.build), 19
BuildSeen (class in changes.models.buildseen), 19

C

Cluster (class in changes.models.node), 20
ClusterNode (class in changes.models.node), 20
Command (class in changes.models.command), 19
Comment (class in changes.models.comment), 19
create_job() (in module changes.jobs.create_job), 23

E

Event (class in changes.models.event), 19

F

FailureReason (class in changes.models.failurereason), 19
FileCoverage (class in changes.models.filecoverage), 19
fire_signal() (in module changes.jobs.signals), 23

I

import_repo() (in module changes.jobs.import_repo), 23
ItemSequence (class in changes.models.itemsequence), 19
ItemStat (class in changes.models.itemstat), 20

J

Job (class in changes.models.job), 20
JobPhase (class in changes.models.jobphase), 20
JobPlan (class in changes.models.jobplan), 20
JobStep (class in changes.models.jobstep), 20

L

LatestGreenBuild (class in changes.models.latest_green_build), 20
LogChunk (class in changes.models.log), 20
LogSource (class in changes.models.log), 20

N

Node (class in changes.models.node), 20

P

Patch (class in changes.models.patch), 21
PhabricatorDiff (class in changes.models.phabricatordiff), 21
Plan (class in changes.models.plan), 21
Project (class in changes.models.project), 21
ProjectOption (class in changes.models.project), 21

R

Repository (class in changes.models.repository), 22
Revision (class in changes.models.revision), 22
run_event_listener() (in module changes.jobs.signals), 23

S

Snapshot (class in changes.models.snapshot), 22
Source (class in changes.models.source), 22
Step (class in changes.models.step), 22
sync_artifact() (in module changes.jobs.sync_artifact), 23
sync_job() (in module changes.jobs.sync_job), 23
sync_job_step() (in module changes.jobs.sync_job_step), 23
sync_repo() (in module changes.jobs.sync_repo), 23

T

Task (class in changes.models.task), 22
TestArtifact (class in changes.models.testartifact), 23
TestCase (class in changes.models.test), 23

U

User (class in changes.models.user), 23