WEST UNIVERSITY OF TIMIȘOARA
FACULTY OF MATHEMATICS AND COMPUTER SCIENCE
STUDY PROGRAM: COMPUTER SCIENCE IN ENGLISH

BACHELOR THESIS

COORDINATOR:
Lect.Univ.Dr.Stelian Mihalaș

GRADUATE:
Nagy Gabriel Alexandru

TIMIŞOARA

2018

WEST UNIVERSITY OF TIMIȘOARA
FACULTY OF MATHEMATICS AND COMPUTER SCIENCE
STUDY PROGRAM: COMPUTER SCIENCE IN ENGLISH

CONTINUOUS INTEGRATION WITH BUILDBOT

COORDINATOR:
Lect.Univ.Dr.Stelian Mihalaș

GRADUATE:
Nagy Gabriel Alexandru

TIMIŞOARA

2018

Abstract

The purpose of this thesis is to shed light on Buildbot as a Continuous Integration tool. Continuous Integration (CI) is the process of automating the build and testing of code every time a team member commits changes to version control. We target to do this by automating most of the delivery process related tasks, freeing the developer to do other software development related tasks.

Using Buildbot, developers cand test their components for possible errors without the risk of committing to version control and breaking the code. Buildbot is an open source project, written in Python on top of the Twisted network programming framework.

In this thesis we will emphasize the extensibility of Buildbot, and how we can use Python and Angular to strengthen the capabilities of this tool, customizing it to best suit our project.

Contents

Chapter 1

Introduction

1.1 Motivation

Generally, every software company, regardless how large or small, uses Continuous Integration principles. Verifying and testing code before delivering and merging with the mainline is of paramount importance, as it saves money as well as time.

While smaller companies might make use of more flexible ways of testing, the larger ones usually run a CI tool such as Jenkins, on which they run builds of every commit that finds its way to the master/trunk branch. This is a widespread and useful practice, albeit narrow, as it only tracks main branches. If a commit happens to break the code, the trunk gets locked and valuable time is spent to fix the regressions. What I will present in this thesis is how developers can use Buildbot as a tool to remotely test their software, without interfering with other people's work. We tell Buildbot the same step sequence the developer uses to test their code, and it does the job for us.

By transferring these tasks to Buildbot, the developer can work on other important issues without worrying about the testing part.

1.2 Contribution

Even though Buildbot is a powerful tool out-of-the-box, it really shines if properly customized and extended. Being an open-source project, it receives frequent contributions from users around the world. The codebase is modular and properly documented, and one can easily begin writing their own extensions and helpers.

Some of Buildbot's default features are:

- run builds on a variety of worker platforms
- arbitrary build process: if you can build it locally, so can Buildbot
- minimal requirements: Python and Twisted
- status delivery through a variety of protocols: web page, email, IRC
- tracks builds in progress, estimates completion time
- flexible configuration by subclassing generic build process classes

- released under the GPL[1] license

The custom implementations that we will go over consist in:

- send an email with the build results to the user using LDAP to search the directory for the user's email address
- parse custom output logs, counting errors and creating a summary of the passed and failed tests
- preferential build steps: do not run this step if the previous failed
- use Flask to deploy a custom web dashboard
- use Angular to deploy an asynchronous, live-reloading, custom web dashboard
- buildbot try script: the developer executes a script in his working directory, and his changes are instantly send to the Buildbot queue to be built
- extend buildbot commands to support additional arguments such as multiple patchfiles

1.3 Structure

Besides presenting the technologies used, before getting into the extensibility of Buildbot, I will also shed light on some of its default implementations to see their purposes.

[1]The **GNU General Public License** (**GPL**), originally written by Richard Stallman of the Free Software Foundation, is a widely used free software license, which guarantees end users the freedom to run, study, share and modify the software.

Chapter 2

Technologies used

2.1 Backend

By definition, Buildbot is a system to automate the compile/test cycle required by most software projects to validate code changes. By automatically rebuilding and testing the tree each time something has changed, build problems are pinpointed quickly, before other developers are inconvenienced by the failure. By running the builds on a variety of platforms, developers who do not have the facilities to test their changes everywhere before checkin will at least know shortly afterwards whether they have broken the build or not. The overall goal is to reduce tree breakage and provide a platform to run tests or code-quality checks that are too annoying or pedantic for any human to waste their time with. Developers get immediate (and potentially public) feedback about their changes, encouraging them to be more careful about testing before checkin.

The Buildbot was written to automate the human process of walking into the office, updating a tree, compiling (and discovering the breakage), finding the developer at fault, and complaining to them about the problem they had introduced. With multiple platforms it was difficult for developers to do the right thing (compile their potential change on all platforms); the buildbot offered a way to help.[10]

2.1.1 Python

The bulk of Buildbot is written in *Python.*

Python is a widely used high-level programming language for general-purpose programming, created by Guido van Rossum and first released in 1991. It is an interpreted language, meaning that it executes its instructions directly and freely, without previously compiling a program into machine-language instructions. The interpreter executes the program directly, translating each statement into a sequence of one or more subroutines already compiled into machine code.

Python has a design philosophy that emphasizes code readability (notably using whitespace indentation to delimit code blocks rather than curly brackets or keywords), and a syntax that allows programmers to express concepts in fewer lines of code than might be used in languages such as *C++* or *Java.*[1] *Python* features a dynamic type system and automatic memory management. It supports multiple programming paradigms, including object-oriented, imperative, functional and procedural, and has a large and comprehensive standard library.[2] Besides being present in the Buildbot backend, we will also encounter it when implementing custom web dashboards, and in *Angular* with *CoffeeScript*, which is a language that cross-compiles to *JavaScript* and has Python-inspired syntax.

Since 2003, Python has consistently ranked in the top ten most popular programming languages in the TIOBE Programming Community Index.[1] As of November 2017, it is the fourth most popular language.[3]

An empirical study found that scripting languages, such as Python, are more productive than conventional languages, such as C and Java, for programming problems involving string manipulation and search in a dictionary, and determined that memory consumption was often "better than Java and not much worse than C or C++".[4]

2.1.2 Twisted

The cornerstone of Buildbot is Twisted, an open-source, event-driven network programming framework written in Python and licensed under the MIT License.

Twisted supports many common transport and application layer protocols, including TCP, UDP, SSL/TLS, HTTP, IMAP, SSH, IRC, and FTP. Like the language in which is written, it is "batteries-included"; Twisted comes with client and server implementations for all of its protocols, as well as utilities that make it easy to configure and deploy production-grade Twisted applications from the command line.

Twisted includes both high- and low-level tools for building performant, cross-platform applications. One can deploy a web or mail server with just a few lines of code, or one can write their own protocol from scratch. At every level, Twisted provides a tested, RFC-conforming, extensible API that makes it possible to rapidly develop powerful network software.[5]

2.1.2.1 Protocol/transport separation

Twisted is designed for complete separation between logical protocols (usually relying on stream-based connection semantics, such as HTTP or POP3) and physical transport layers supporting such stream-based semantics (such as files, sockets or SSL libraries). Connection between a logical protocol and a transport layer happens at the last possible moment — just before information is passed into the logical protocol instance. The logical protocol is informed of the transport layer instance, and can use it to send messages back and to check for the peer's identity. Note that it is still possible, in protocol code, to deeply query the transport layer on transport issues (such as checking a client-side SSL certificate). Naturally, such protocol code will fail (raise an exception) if the transport layer does not support such semantics.

2.1.2.2 The Deferred API

Central to the Twisted application model is the concept of a deferred (elsewhere called a future). A deferred is an instance of a class designed to receive and process a result which has not been computed yet, for example because it is based on data from a remote peer. Deferreds can be passed around, just like regular objects, but cannot be asked for their value. Each deferred supports a callback chain. When the deferred

[1]A measure of popularity of programming languages, created and maintained by the TIOBE Company based in Eindhoven, Netherlands. TIOBE stands for "The Importance of Being Earnest", taken from the name of a comedy play written by Oscar Wilde at the end of the nineteenth century.

gets the value, it is passed to the functions on the callback chain, with the result of each callback becoming the input for the next. Deferreds make it possible to arrange to operate on the result of a function call before its value has become available.

For example, if a deferred returns a string from a remote peer containing an IP address in quad format, a callback can be attached to translate it into a 32-bit number. Any user of the deferred can now treat it as a deferred returning a 32-bit number. This, and the related ability to define ”errbacks” (callbacks which are called as error handlers), allows code to specify in advance what to do when an asynchronous event occurs, without stopping to wait for the event. In non-event-driven systems, for example using threads, the operating system incurs premature and additional overhead organizing threads each time a blocking call is made.

2.1.2.3 Thread support

Twisted supports an abstraction over raw threads — using a thread as a deferred source. Thus, a deferred is returned immediately, which will receive a value when the thread finishes. Callbacks can be attached which will run in the main thread, thus alleviating the need for complex locking solutions. A prime example of such usage, which comes from Twisted's support libraries, is using this model to call into databases. The database call itself happens on a foreign thread, but the analysis of the result happens in the main thread.[7]

2.1.2.4 Example: A TCP Echo Server and Client

A basic implementation for a simple TCP echo server and client pair can be seen below. The server's job is to listen for TCP connections on a particular port and echo back anything it receives. The client's job is to connect to the server, send it a message, receive a response, and terminate the connection.[6]

```python
from twisted.internet import protocol, reactor

class Echo(protocol.Protocol):
   def dataReceived(self, data):
      self.transport.write(data)

class EchoFactory(protocol.Factory):
   def buildProtocol(self, addr):
      return Echo()

reactor.listenTCP(8000, EchoFactory())
reactor.run()
```

Listing 2.1: *echoserver.py*

```python
from twisted.internet import reactor, protocol

class EchoClient(protocol.Protocol):
   def connectionMade(self):
      self.transport.write("Hello, world!")
```

```
    def dataReceived(self, data):
        print "Server said:", data
        self.transport.loseConnection()

class EchoFactory(protocol.ClientFactory):
    def buildProtocol(self, addr):
        return EchoClient()

    def clientConnectionFailed(self, connector, reason):
        print "Connection failed."
        reactor.stop()

    def clientConnectionLost(self, connector, reason):
        print "Connection lost."
        reactor.stop()

reactor.connectTCP("localhost", 8000, EchoFactory())
reactor.run()
```

Listing 2.2: *echoclient.py*

To test the scripts, the server should be run in a first terminal using `python echoserver.py`. This will start a TCP server listening for connections on port 8000. The client should be run in a second terminal with `python echoclient.py`.

A sample command sequence would be:

```
$ python echoserver.py # in terminal 1

$ python echoclient.py # in terminal 2
Server said: Hello, world!
Connection lost.
```

Listing 2.3: Command sequence example

2.1.3 Database

As of version 0.8.0, Buildbot has used a database as part of its storage backend. All access to the Buildbot database is mediated by database connector classes. These classes provide a functional, asynchronous interface to other parts of Buildbot, and encapsulate the database-specific details in a single location in the codebase.

The connectors all use *SQLAlchemy Core* to achieve (almost) database-independent operation. Note that the *SQLAlchemy ORM* is not used in Buildbot. Database queries are carried out in threads, and report their results back to the main thread via Twisted Deferreds. The database schema is maintained with *SQLAlchemy-Migrate*. This package handles the details of upgrading users between different schema versions.[11]

In the default configuration Buildbot uses a file-based *SQLite* database, stored in the `state.sqlite` file of the master's base directory.[12]

SQLite is a relational database management system (DBMS) contained in a C programming library. In contrast to many other database management systems, SQLite is not a client–server database engine. Rather, it is embedded into the end program. SQLite is ACID-compliant[2] and implements most of the SQL standard, using a dynamically and weakly typed SQL syntax that does not guarantee the domain integrity.[8] SQLite is a popular choice as embedded database software for local/client storage in application software such as web browsers. It is arguably the most widely deployed database engine, as it is used today by several widespread browsers, operating systems, and embedded systems (such as mobile phones), among others. SQLite has bindings to many programming languages, including Python. Among its most popular users are browsers like *Google Chrome, Safari* and *Mozilla Firefox* which stores a variety of configuration data (bookmarks, cookies, contacts etc.) inside internally managed SQLite databases.[9]

Although Buildbot's default database is SQLite, it also supports MySQL and PostgreSQL.

2.2 Web interface

In previous releases, the web part of buildbot was implemented using a WebStatus plugin. That has since become deprecated, and as of Buildbot 0.9.0, the built-in web server supersedes it.

The client side of the web UI is written in JavaScript and based on the Angular framework and concepts.

Being a Single Page Application, all Buildbot pages are loaded from the same path, at the master's base URL. A JavaScript technique is used to avoid reloading the whole page over HTTP when the users changes the URI or clicks a link.

2.2.1 Flask

Flask is a BSD licensed micro web framework written in Python, based on *Werkzeug*, a powerful WSGI utility library for Python, and *Jinja2*, a full featured template engine.[70] We will make use of Flask's modular design to quickly deploy web dashboards for Buildbot.

2.2.2 Angular

Angular is a framework for building client applications in HTML and either JavaScript or a language like TypeScript that compiles to JavaScript (in Buildbot's case, CoffeeScript, because of its similarities to Python).[13]

The framework works by first reading the HTML page, which has additional custom tag attributes embedded into it. Angular interprets those attributes as directives to bind input or output parts of the page to a model that is represented by standard JavaScript variables. The values of those JavaScript variables can be manually set within the code, or retrieved from static or dynamic JSON resources.[14]

Angular is built around the belief that declarative code is better than imperative when it comes to building UIs and wiring software components together, while imperative code is excellent for expressing business logic.

[2]Acronym for *Atomicity, Consistency, Isolation, Durability.* A set of properties of database transactions intended to guarantee validity even in the event of errors, power failures, etc.

Angular's design goals include:

- decoupling DOM manipulation from app logic, improving the testability of the code
- decoupling the client side of an application from the server side, allowing development work to progress in paralel
- providing guidance through the entire journey of building an app, from designing the UI, through writing the business logic, to testing
- making common tasks trivial and difficult tasks possible[15]

2.2.3 NodeJS

Node.js (Node) is an open source development platform for executing JavaScript code server-side. Node is useful for developing applications that require a persistent connection from the browser to the server and is often used for real-time applications such as chat, news feeds and web push notifications.

Node.js is intended to run on a dedicated HTTP server and to employ a single thread with one process at a time. Node.js applications are event-based and run asynchronously. Code built on the Node platform does not follow the traditional model of receive, process, send, wait, receive. Instead, Node processes incoming requests in a constant event stack and sends small requests one after the other without waiting for responses.

This is a shift away from mainstream models that run larger, more complex processes and run several threads concurrently, with each thread waiting for its appropriate response before moving on.

One of the major advantages of Node.js, according to its creator Ryan Dahl, is that it does not block input/output (I/O). Some developers are highly critical of Node.js and point out that if a single process requires a significant number of CPU cycles, the application will block and that the blocking can crash the application. Proponents of the Node.js model claim that CPU processing time is less of a concern because of the high number of small processes that Node code is based on.[16]

2.2.4 npm

npm is a package manager for the JavaScript programming language. It is the default package manager for the JavaScript runtime environment *Node.js*. It consists of a command line client, also called *npm*, and an online database of public and paid-for private packages, called the npm registry. The registry is accessed via the client, and the available packages can be browsed and searched via the npm website.

npm is included as a recommended feature in Node.js installer.[17] npm consists of a command line client that interacts with a remote registry. It allows users to consume and distribute JavaScript modules that are available on the registry.[18] Packages on the registry are in CommonJS format and include a metadata file in JSON format.[19] Over 477,000 packages are available on the main npm registry.[20] The registry has no vetting process for submission, which means that packages found there can be low quality, insecure, or malicious.[19] Instead, npm relies on user reports to take down packages if they violate policies by being insecure, malicious or low quality.[21] npm exposes statistics including number of downloads and number of depending packages to assist developers in judging the quality of packages.[22]

npm can manage packages that are local dependencies of a particular project, as well as globally-installed JavaScript tools.[23] When used as a dependency manager for a local project, npm can install, in one command, all the dependencies of a project through the `package.json` file.[24] In the package.json file, each dependency can specify a range of valid versions using the semantic versioning scheme, allowing developers to auto-update their packages while at the same time avoiding unwanted breaking changes.[25] npm also provides version-bumping tools for developers to tag their packages with a particular version.[26]

There are a number of alternatives to *npm* for installing modular JavaScript, most popular of which is *Yarn*, which was open-sourced by Facebook in October 2016.[27] One of the main reasons for this development was that *npm* broke down on Facebook's continuous integration environments, due to sandboxing and no internet access. Yarn replaces the existing workflow for the npm client while remaining compatible with the npm registry.

In the Node ecosystem, dependencies get placed within a `node_modules` directory in your project. However, this file structure can differ from the actual dependency tree as duplicate dependencies are merged together. The npm client installs dependencies into the `node_modules` directory non-deterministically. This means that based on the order dependencies are installed, the structure of a `node_modules` directory could be different from one person to another. These differences can cause "works on my machine" bugs that take a long time to hunt down.

Yarn resolves these issues around versioning and non-determinism by using lockfiles and an install algorithm that is deterministic and reliable. These lockfiles lock the installed dependencies to a specific version, and ensure that every install results in the exact same file structure in `node_modules` across all machines. The written lockfile uses a concise format with ordered keys to ensure that changes are minimal and review is simple.[28]

2.2.5 Gulp

gulp.js is an open-source JavaScript toolkit by Fractal Innovations and the open source community at GitHub, used as a streaming build system in front-end web development.[29]

It is a task runner built on *Node.js* and *npm*, used for automation of time-consuming and repetitive tasks involved in web development like minification, concatenation, cache busting, unit testing, linting, optimization, etc.[32]

gulp is a build tool in JavaScript built on node streams. These streams facilitate the connection of file operations through pipelines.[30] gulp reads the file system and pipes the data at hand from its one single-purposed plugin to other through the .pipe() operator, doing one task at a time. The original files are not affected until all the plugins are processed. It can be configured either to modify the original files or to create new ones. This grants the ability to perform complex tasks through linking its numerous plugins. The users can also write their own plugins to define their own tasks.[31] Unlike other task runners that run tasks by configuration, gulp requires knowledge of JavaScript and coding to define its tasks. gulp is a build system which means apart from running tasks, it is also capable of copying files from one location to another, compiling, deploying, creating notifications, unit testing, linting, etc.[29]

Task-runners like gulp and grunt are built on Node.js rather than npm, because the basic npm scripts are inefficient when executing multiple tasks. Even though some developers prefer npm scripts because they can be simple and easy to implement, there are numerous ways where gulp and grunt seem to have an advantage over each other and the default provided scripts.[33] Grunt runs tasks by transforming files and saves as new ones in temporary folders and the output of one task is taken as input for another and so on until the output reaches the destination folder. This involves a lot of I/O calls and creation of many temporary files. Whereas gulp streams through the file system and does not require any of these temporary locations decreasing the number of I/O calls thus, improving performance.[34] Grunt uses configuration files to perform tasks whereas gulp requires its build file to be coded. In grunt, each plugin needs to be configured to match its input location to the previous plugin's output. In gulp, the plugins are automatically pipe-lined.[30]

The gulp tasks are run from a **Command Line Interface** (CLI)[33] shell and require `package.json` and `gulpfile.js` (simply gulpfile) in the project root directory. The gulpfile is where plugins are loaded and tasks are defined. First, all the necessary modules are loaded and then tasks are defined in the gulpfile. All the necessary plugins specified in the gulpfile are installed into the devDependencies.[24] The default task is run with `gulp`. Individual tasks can be defined by gulp.task and are run by `gulp <task> <othertask>`.[35] Complex tasks are defined by chaining the plugins with the help of the `.pipe()` operator.[36]

2.2.6 CoffeeScript

CoffeeScript is a programming language that transcompiles to *JavaScript*. It adds syntactic sugar inspired by Ruby, Python and Haskell in an effort to enhance JavaScript's brevity and readability.[37] Specific additional features include list comprehension and pattern matching.

Almost everything is an expression in CoffeeScript, for example `if`, `switch` and `for` expressions (which have no return value in JavaScript) return a value. As in Perl, these control statements also have postfix versions; for example, `if` can also be written after the conditional statement.

Many unnecessary parentheses and braces can be omitted; for example, blocks of code can be denoted by indentation instead of braces, function calls are implicit, and object literals are often detected automatically.

CoffeeScript was chosen for Buildbot development because of its familiarities with Python. As most of its developers are Python experts, they found it helpful to code the frontend using a similar syntax as the backend.[42]

As of December 2017, switching the buildbot frontend codebase from CoffeeScript to TypeScript or another more-modern language is being considered.[41]

2.2.6.1 Code snippets

To compute the greatest common divisor of two integers using the euclidean algorithm, in JavaScript one usually needs a `while` loop:

```
gcd = (x, y) => {
  do {
    z = x % y
    x = y
    y = z
  } while (y !== 0)
  return x
}
```

Listing 2.4: *loop.js*

Whereas in CoffeeScript one can use `until` and pattern-matching instead:

```
gcd = (x, y) ->
  [x, y] = [y, x%y] until y is 0
  x
```

Listing 2.5: *loop.coffee*

A linear search can be implemented in one line using the `when` keyword:

```
names = ["Long", "John", "Silver"]
linearSearch = (searchName) -> alert(name) for name in names when name
    is searchName
```

Listing 2.6: *linearSearch.coffee*

Ruby-style string interpolation is included in CoffeeScript. Double-quoted strings allow for interpolated values using `#{...}` and single-quoted strings are treated as literals:[38]

```
muppet = "Beeker"
favorite = "My favorite muppet is #{muppet}!"

# => "My favorite muppet is Beeker!"
```

Listing 2.7: *interpolation.coffee*

CoffeeScript allows us to chain two comparisons together in a form that more closely matches the way a mathematician would write it:[39]

```
maxDwarfism = 147
minAcromegaly = 213

height = 180

normalHeight = maxDwarfism < height < minAcromegaly
# => true
```

Listing 2.8: *comparingRanges.coffee*

A simple way to integrate small snippets of JavaScript code into a CoffeeScript codebase, without converting its syntax would be to wrap the JavaScript with backticks:[40]

```
`function greet(name) {
return "Hello "+name;
}`
```

```
# Back to CoffeeScript
greet "Coffee"
# => "Hello Coffee"
```

Listing 2.9: *embedJS.coffee*

2.2.7 pug

Formerly known as Jade (a registered trademark, and as a result a rename was needed), *pug* is a high performance and feature-rich templating engine. Simply put, *pug* is a clean, whitespace/indentation sensitive syntax for simplifying and escaping the archaic syntax of HTML.

Just like SASS, Pug is a prepocessor and, as such it facilitates accomplishing tasks like wrapping away repetitive work by providing features not available in plain HTML. It provides the ability to write dynamic and reusable HTML documents, it's an open source HTML templating language for Node.js (server-side JavaScript), totally free to use and provides fast and easy HTML.[43]

A basic syntax comparison can be seen below:

```
<html>
  <head>
    <title>This is my first Pug file</title>
  </head>
  <body>
    <header>
      <p>My soon to be menu</p>
    </header>
    <section>
      <p>This is a post about Pug template engine</p>
    </section>
    <footer>
      lots of copyrights
    </footer>
  </body>
</html>
```

Listing 2.10: *example.html*

```
html
    head
        title This is my first Pug file
    body
        header
            p My soon to be menu
        section
            p This is a post about Pug template engine
        footer
            lots of copyrights
```

Listing 2.11: *example.pug*

Chapter 3

Default implementations

3.1 Concepts and terminology

In its 15 years of existence, buildbot has defined some basic concepts that need to be understood if one wants to configure and extend the platform.

3.1.1 Source Stamps

Source code comes from repositories, provided by version control systems. Repositories are generally identified by URLs, e.g., `git://github.com/buildbot/buildbot.git`.

In these days of distributed version control systems, the same *codebase* may appear in multiple repositories. For example, `https://github.com/mozilla/mozilla-central` and `http://hg.mozilla.org/mozilla-release` both contain the Firefox codebase, although not exactly the same code.

Many *projects* are built from multiple codebases. For example, a company may build several applications based on the same core library. The “app” codebase and the “core” codebase are in separate repositories, but are compiled together and constitute a single project. Changes to either codebase should cause a rebuild of the application.

Most version control systems define some sort of *revision* that can be used (sometimes in combination with a *branch*) to uniquely specify a particular version of the source code.

To build a project, Buildbot needs to know exactly which version of each codebase it should build. It uses a *source stamp* to do so for each codebase; the collection of sourcestamps required for a project is called a *source stamp set.*[44]

3.1.2 Changes

3.1.2.1 Who

Each `Change` has a `who` attribute, which specifies which developer is responsible for the change. This is a string which comes from a namespace controlled by the VC repository. Frequently this means it is a username on the host which runs the repository, but not all VC systems require this. Each `StatusNotifier` will map the `who` attribute into something appropriate for their particular means of communication: an email address, an IRC handle, etc.[45]

3.1.2.2 Files

It also has a list of `files`, which are just the tree-relative filenames of any files that were added, deleted, or modified for this `Change`. These filenames are used by the `fileIsImportant` function (in the scheduler) to decide whether it is worth triggering a new build or not, e.g. the function could use the following function to only run a build if a C file were checked in:[46]

```
def has_C_files(change):
    for name in change.files:
        if name.endswith(".c"):
            return True
    return False
```

Listing 3.1: *has_C_files.py*

3.1.2.3 Comments

The Change also has a `comments` attribute, which is a string containing any checkin comments.[47]

3.1.2.4 Project

The `project` attribute of a change or source stamp describes the project to which it corresponds, as a short human-readable string. This is useful in cases where multiple independent projects are built on the same buildmaster. In such cases, it can be used to control which builds are scheduled for a given commit, and to limit status displays to only one project.[48]

3.1.2.5 Repository

This attribute specifies the repository in which this change occurred. In the case of DVCS's, this information may be required to check out the committed source code. However, using the repository from a change has security risks: if Buildbot is configured to blindly trust this information, then it may easily be tricked into building arbitrary source code, potentially compromising the workers and the integrity of subsequent builds.[49]

3.1.2.6 Codebase

This attribute specifies the codebase to which this change was made. As described in the Source Stamps section, multiple repositories may contain the same codebase. A change's codebase is usually determined by the `codebaseGenerator` configuration. By default the codebase is '' (empty); this value is used automatically for single-codebase configurations.[50] For our implementation we will be using the default option for this attribute.

3.1.2.7 Revision

Each Change can have a `revision` attribute, which describes how to get a tree with a specific state: a tree which includes this Change (and all that came before it) but none that come after it. If this information is unavailable, the `revision` attribute will be `None`. These revisions are provided by the `ChangeSource`.

Revisions are always strings.[51]

3.1.2.8 Branches

The Change might also have a `branch` attribute. This indicates that all of the Change's files are in the same named branch. The schedulers get to decide whether the branch should be built or not.[53]

3.1.2.9 Change Properties

A Change may have one or more properties attached to it, usually specified through the Force Build form or `sendchange`. These are useful for custom buildbot implementations.[52]

3.1.3 Schedulers

Each Buildmaster has a set of scheduler objects, each of which gets a copy of every incoming `Change`. The Schedulers are responsible for deciding when `Builds` should be run. Some Buildbot installations might have a single scheduler, while others may have several, each for a different purpose.

For example, a *quick* scheduler might exist to give immediate feedback to developers, hoping to catch obvious problems in the code that can be detected quickly. These typically do not run the full test suite, nor do they run on a wide variety of platforms. They also usually do a VC update rather than performing a brand-new checkout each time.

A separate *full* scheduler might run more comprehensive tests, to catch more subtle problems. configured to run after the quick scheduler, to give developers time to commit fixes to bugs caught by the quick scheduler before running the comprehensive tests. This scheduler would also feed multiple `Builders`.

Many schedulers can be configured to wait a while after seeing a source-code change - this is the *tree stable timer*. The timer allows multiple commits to be "batched" together. This is particularly useful in distributed version control systems, where a developer may push a long sequence of changes all at once. To save resources, it's often desirable only to test the most recent change.

Schedulers can also filter out the changes they are interested in, based on a number of criteria. For example, a scheduler that only builds documentation might skip any changes that do not affect the documentation. Schedulers can also filter on the branch to which a commit was made.

There is some support for configuring dependencies between builds - for example, you may want to build packages only for revisions which pass all of the unit tests. This support is under active development in Buildbot, and is referred to as "build coordination".

Periodic builds (those which are run every N seconds rather than after new Changes arrive) are triggered by a special `Periodic` scheduler.

Each scheduler creates and submits `BuildSet` objects to the `BuildMaster`, which is then responsible for making sure the individual `BuildRequests` are delivered to the target `Builders`.

Scheduler instances are activated by placing them in the `schedulers` list in the buildmaster config file. Each scheduler must have a unique name.[54]

3.1.4 BuildSets

A `BuildSet` is the name given to a set of `Build`s that all compile/test the same version of the tree on multiple `Builders`. In general, all these component `Build`s will perform the same sequence of `Step`s, using the same source code, but on different platforms or against a different set of libraries.

The `BuildSet` is tracked as a single unit, which fails if any of the component `Build`s have failed, and therefore can succeed only if *all* of the component `Build`s have succeeded. There are two kinds of status notification messages that can be emitted for a `BuildSet`: the `firstFailure` type (which fires as soon as we know the `BuildSet` will fail), and the `Finished` type (which fires once the `BuildSet` has completely finished, regardless of whether the overall set passed or failed).

A `BuildSet` is created with set of one or more *source stamp* tuples of `(branch, revision, changes, patch)`, some of which may be `None`, and a list of `Builder`s on which it is to be run. They are then given to the BuildMaster, which is responsible for creating a separate `BuildRequest` for each `Builder`.

The buildmaster is responsible for turning the `BuildSet` into a set of `BuildRequest` objects and queueing them on the appropriate `Builders`.[55]

3.1.5 BuildRequests

A `BuildRequest` is a request to build a specific set of source code (specified by one ore more source stamps) on a single `Builder`. Each `Builder` runs the `BuildRequest` as soon as it can (i.e. when an associated worker becomes free). `BuildRequest`s are prioritized from oldest to newest, so when a worker becomes free, the `Builder` with the oldest `BuildRequest` is run.

The `BuildRequest` contains one `SourceStamp` specification per codebase. The actual process of running the build (the series of `Step`s that will be executed) is implemented by the `Build` object. In the future this might be changed, to have the `Build` define *what* gets built, and a separate `BuildProcess` (provided by the Builder) to define *how* it gets built.

The `BuildRequest` may be mergeable with other compatible `BuildRequest`s. Builds that are triggered by incoming `Change`s will generally be mergeable. Builds that are triggered by user requests are generally not, unless they are multiple requests to build the *latest sources* of the same branch. A merge of buildrequests is performed per codebase, thus on changes having the same codebase.[56]

3.1.6 Builders

The Buildmaster runs a collection of `Builder`s, each of which handles a single type of build (e.g. full versus quick), on one or more workers. `Builder`s serve as a kind of queue for a particular type of build. Each `Builder` gets a separate column in the waterfall display. In general, each `Builder` runs independently (although various kinds of interlocks can cause one `Builder` to have an effect on another).

Each builder is a long-lived object which controls a sequence of `Builds`. Each `Builder` is created when the config file is first parsed, and lives forever (or rather until it is removed from the config file). It mediates the connections to the workers that do all the work, and is responsible for creating the `Build` objects.

Each builder gets a unique name, and the path name of a directory where it gets to do all its work (there is a buildmaster-side directory for keeping status information, as well as a worker-side directory where the actual checkout/compile/test commands are executed).

A builder also has a `BuildFactory`, which is responsible for creating new `Build` instances: because the `Build` instance is what actually performs each build, choosing the `BuildFactory` is the way to specify what happens each time a build is done.[57]

3.1.7 Builds

A build is a single compile or test run of a particular version of the source code, and is comprised of a series of steps. It is ultimately up to you what constitutes a build, but for compiled software it is generally the checkout, configure, make, and make check sequence. For interpreted projects like Python modules, a build is generally a checkout followed by an invocation of the bundled test suite.

A `BuildFactory` describes the steps a build will perform. The builder which starts a build uses its configured build factory to determine the build's steps.[59]

3.1.8 Workers

Each builder is associated with one of more `Workers`. A builder which is used to perform Mac OS X builds (as opposed to Linux or Solaris builds) should naturally be associated with a Mac worker.

If multiple workers are available for any given builder, you will have some measure of redundancy: in case one worker goes offline, the others can still keep the `Builder` working. In addition, multiple workers will allow multiple simultaneous builds for the same `Builder`, which might be useful if you have a lot of forced or `try` builds taking place.

If you use this feature, it is important to make sure that the workers are all, in fact, capable of running the given build. The worker hosts should be configured similarly, otherwise you will spend a lot of time trying (unsuccessfully) to reproduce a failure that only occurs on some of the workers and not the others. Different platforms, operating systems, versions of major programs or libraries, all these things mean you should use separate Builders.[58]

3.1.9 Users

Buildbot has a somewhat limited awareness of *users*. It assumes the world consists of a set of developers, each of whom can be described by a couple of simple attributes. These developers make changes to the source code, causing builds which may succeed or fail.

Users also may have different levels of authorization when issuing Buildbot commands, such as forcing a build from the web interface or from an IRC channel.

Each developer is primarily known through the source control system. Each `Change` object that arrives is tagged with a `who` field that typically gives the account name (on the repository machine) of the user responsible for that change. This string is displayed on the HTML status pages and in each `Build`'s *blamelist*.

To do more with the User than just refer to them, this username needs to be mapped into an address of some sort. The responsibility for this mapping is left up to the status module which needs the address. In the future, the responsibility for managing users will be transferred to User Objects.

The `who` fields in `git` Changes are used to create *User Objects*, which allow for more control and flexibility in how Buildbot manages users.[60]

3.1.10 Build Properties

Each build has a set of *Build Properties*, which can be used by its build steps to modify their actions. These properties, in the form of key-value pairs, provide a general framework for dynamically altering the behavior of a build based on its circumstances.

Properties form a simple kind of variable in a build. Some properties are set when the build starts, and properties can be changed as a build progresses – properties set or changed in one step may be accessed in subsequent steps. Property values can be numbers, strings, lists, or dictionaries - basically, anything that can be represented in JSON.

Properties are very flexible, and can be used to implement all manner of functionality. Here are some examples:

Most Source steps record the revision that they checked out in the `got_revision` property. A later step could use this property to specify the name of a fully-built tarball, dropped in an easily-accessible directory for later testing.

Some projects want to perform nightly builds as well as building in response to committed changes. Such a project would run two schedulers, both pointing to the same set of builders, but could provide an `is_nightly` property so that steps can distinguish the nightly builds, perhaps to run more resource-intensive tests.

Some projects have different build processes on different systems. Rather than create a build factory for each worker, the steps can use worker properties to identify the unique aspects of each worker and adapt the build process dynamically.[61]

3.1.11 Data API

The data layer combines access to stored state and messages, ensuring consistency between them, and exposing a well-defined API that can be used both internally and externally. Using caching and the clock information provided by the db and mq layers, this layer ensures that its callers can easily receive a dump of current state plus changes to that state, without missing or duplicating messages.

3.1.11.1 Sections

The data API is divided into four sections:

- getters - fetching data from the database API;
- subscriptions - subscribing to messages from the mq layer;

- control - allows state to be changed in specific ways by sending appropriate messages (e.g., stopping a build);
- updates - direct updates to state appropriate messages.

The getters and subscriptions are exposed everywhere. Access to the control section should be authenticated at higher levels, as the data layer does no authentication. The updates section is for use only by the process layer.

The interfaces for all sections but the updates sections are intended to be language-agnostic. That is, they should be callable from JavaScript via HTTP, or via some other interface added to Buildbot after the fact.

3.1.11.2 Updates

The updates section defines a free-form set of methods that Buildbot's process implementation calls to update data. Most update methods both modify state via the database API and send a message via the message queue API. Some are simple wrappers for these APIs, while others contain more complex logic, e.g., building a source stamp set for a collection of changes. This section is the proper place to put common functionality, e.g., rebuilding builds or assembling buildsets.

3.1.11.3 Data Model

The data API enforces a strong and well-defined model on Buildbot's data. This model is influenced by REST, in the sense that it defines resources, representations for resources, and identifiers for resources. For each resource type, the API specifies

- the attributes of the resource and their types (e.g., changes have a string specifying their project);
- the format of links to other resources (e.g., buildsets to sourcestamp sets);
- the paths relating to the resource type;
- the format of routing keys for messages relating to the resource type;
- the events that can occur on that resource (e.g., a buildrequest can be claimed);
- options and filters for getting resources.

Some resource type attributes only appear in certain formats, as noted in the documentation for the resource types. In general, messages do not include any optional attributes, nor links.

Paths are given here separated by slashes, with key names prefixed by `:` and described below. Similarly, message routing keys given here are separated by dots, with key names prefixed by `$`. The translation to tuples and other formats should be obvious.

All strings in the data model are unicode strings.[63]

3.2 Web interface

One of the goals of the 'nine' project is to rework Buildbot's web services to use a more modern, consistent design and implement UI features in client-side JavaScript instead of server-side Python.

The rationale behind this is that a client side UI relieves pressure on the server while being more responsive for the user. The web server only concentrates on serving data via a REST interface wrapping the Data API. This removes a lot of sources of latency where, in previous versions, long synchronous calculations were made on the server to generate complex pages.

Another big advantage is live updates of status pages, without having to poll or reload. The new system uses Comet techniques in order to relay Data API events to connected clients.

Finally, making web services an integral part of Buildbot, rather than a status plugin, allows tighter integration with the rest of the application.

The `www` service exposes three pieces via HTTP:

- A REST interface wrapping Data API;
- HTTP-based messaging protocols wrapping the Messaging and Queues interface;
- Static resources implementing the client-side UI.

The REST interface is a very thin wrapper: URLs are translated directly into Data API paths, and results are returned directly, in JSON format, based on the JSON API. Control calls are handled with a simplified form of JSONRPC 2.0.

The message interface is also a thin wrapper around Buildbot's MQ mechanism. Clients can subscribe to messages, and receive copies of the messages, in JSON, as they are received by the buildmaster.

The client-side UI is an AngularJS application. Buildbot uses the Python setuptools entry-point mechanism to allow multiple packages to be combined into a single client-side experience. This allows frontend developers and users to build custom components for the web UI without hacking Buildbot itself.

Python development and AngularJS development are very different processes, requiring different environment requirements and skillsets. To maximize hackability, Buildbot separates the two cleanly. An experienced AngularJS hacker should be quite comfortable in the www/ directory, with a few exceptions described below. Similarly, an experienced Python hacker can simply download the pre-built web UI and never venture near the www/ directory.

URLs

The Buildbot web interface is rooted at its base URL, as configured by the user. It is entirely possible for this base URL to contain path components, e.g., `http://build.example.org/bu` if hosted behind an HTTP proxy. To accomplish this, all URLs are generated relative to the base URL.

Overall, the space under the base URL looks like this:

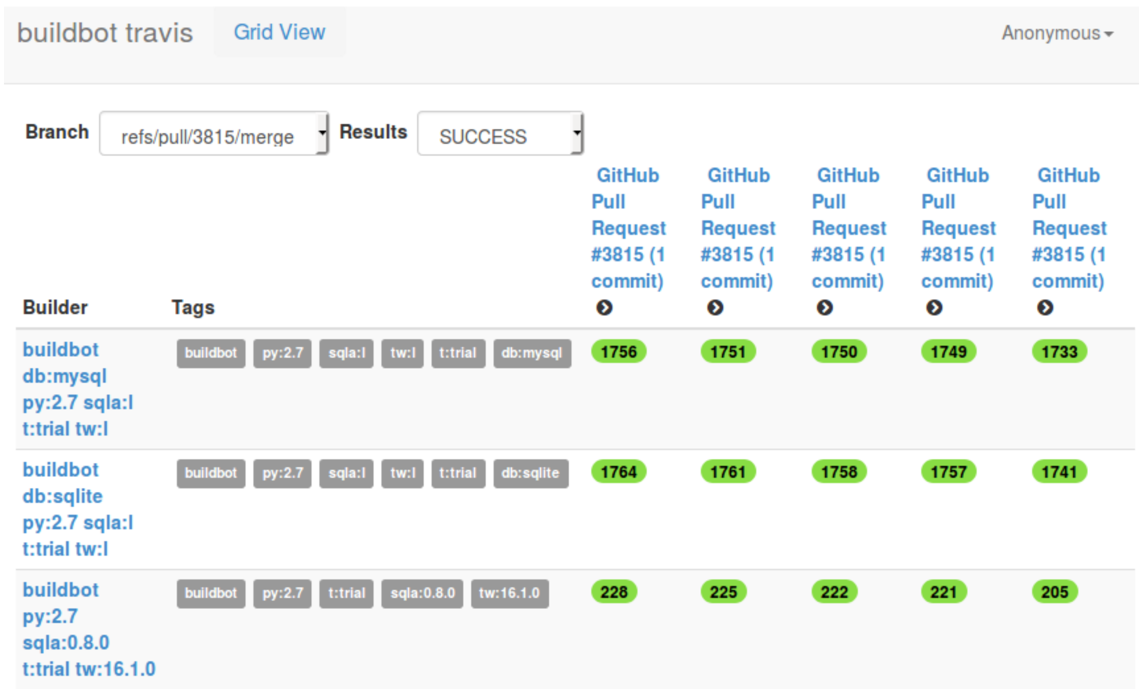
- `/` – The HTML document that loads the UI

- `/api/v{version}` – The root of the REST APIs, each versioned numerically. Users should, in general, use the latest version.
- `/ws` – The WebSocket endpoint to subscribe to messages from the mq system.
- `/sse` – The server sent event endpoint where clients can subscribe to messages from the mq system.[62]

3.2.1 Grid view

Grid view shows the whole buildbot activity arranged by builders vertically and changes horizontally. It is equivalent to Buildbot Eight's grid view.

By default, changes on all branches are displayed but only one branch may be filtered by the user. Builders can also be filtered by tags. This feature is similar to the one in the builder list.[64]



Branch refs/pull/3815/merge Results SUCCESS

Builder	Tags	GitHub Pull Request #3815 (1 commit)	GitHub Pull Request #3815 (1 commit)	GitHub Pull Request #3815 (1 commit)	GitHub Pull Request #3815 (1 commit)	GitHub Pull Request #3815 (1 commit)
buildbot db:mysql py:2.7 sqla:l t:trial tw:l	buildbot py:2.7 sqla:l tw:l t:trial db:mysql	1756	1751	1750	1749	1733
buildbot db:sqlite py:2.7 sqla:l t:trial tw:l	buildbot py:2.7 sqla:l tw:l t:trial db:sqlite	1764	1761	1758	1757	1741
buildbot py:2.7 sqla:0.8.0 t:trial tw:16.1.0	buildbot py:2.7 t:trial sqla:0.8.0 tw:16.1.0	228	225	222	221	205

Figure 3.1: Sample grid view

3.2.2 Waterfall view

Waterfall shows the whole buildbot activity in a vertical time line. Builds are represented with boxes whose height vary according to their duration. Builds are sorted by builders in the horizontal axes, which allow you to see how builders are scheduled together.[66]

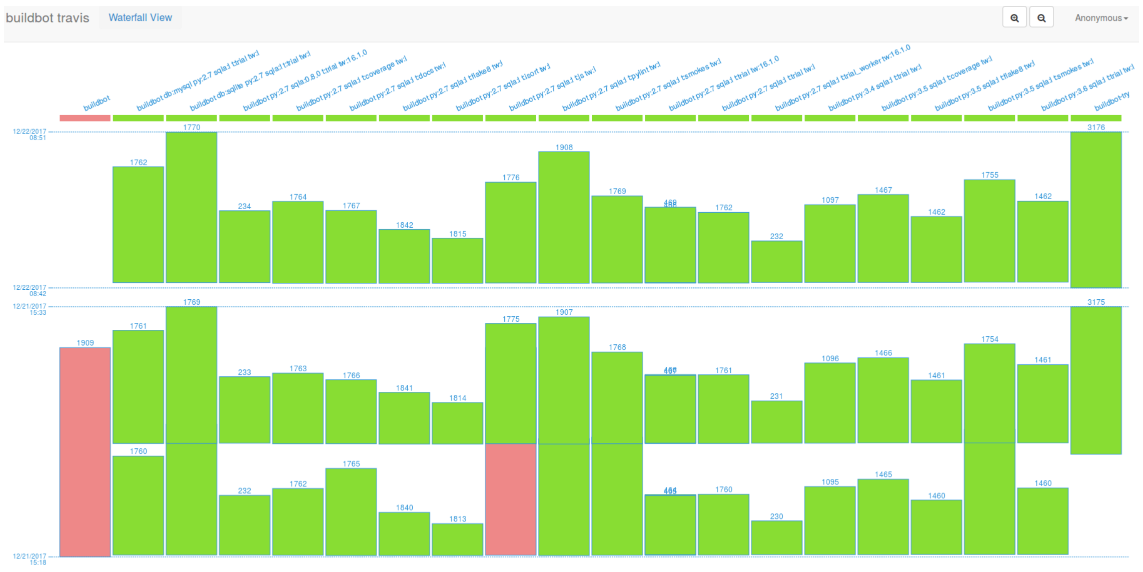
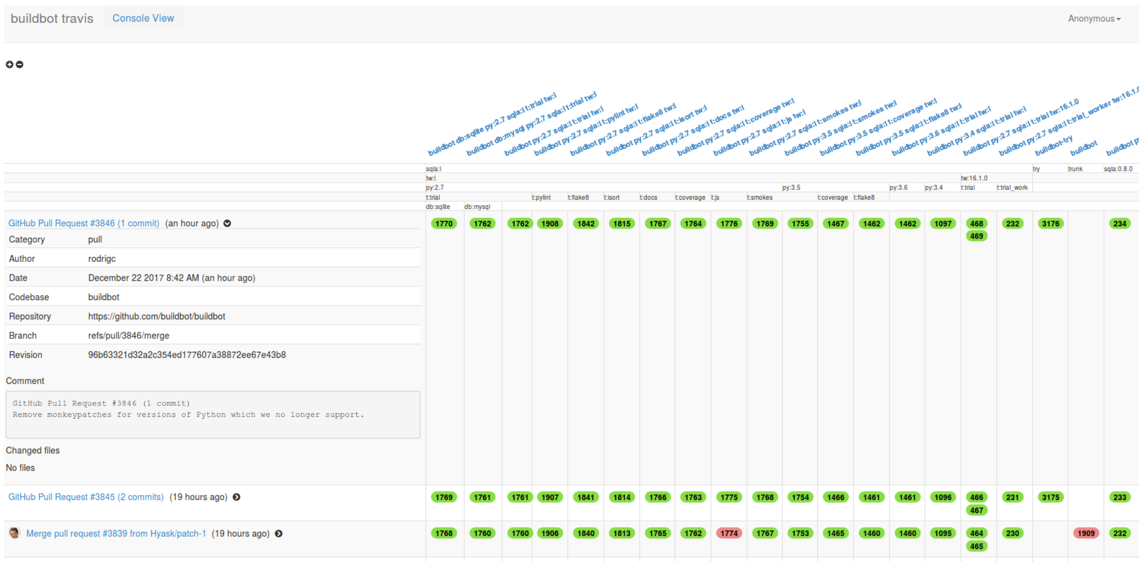


Figure 3.2: Sample waterfall view

3.2.3 Console view

Console view shows the whole buildbot activity arranged by changes as discovered by Change Sources vertically and builders horizontally. If a builder has no build in the current time range, it will not be displayed. If no change is available for a build, then it will generate a fake change according to the `got_revision` property.

Console view will also group the builders by tags. When there are several tags defined per builders, it will first group the builders by the tag that is defined for most builders. Then given those builders, it will group them again in another tag cluster.[65]



GitHub Pull Request #3846 (1 commit) (an hour ago)

Category	pull
Author	rodrigc
Date	December 22 2017 8:42 AM (an hour ago)
Codebase	buildbot
Repository	https://github.com/buildbot/buildbot
Branch	refs/pull/3846/merge
Revision	96b63321d32a2c354ed177607a38872ee67e43b8

Comment

```
GitHub Pull Request #3846 (1 commit)
Remove monkeypatches for versions of Python which we no longer support.
```

Changed files

No files

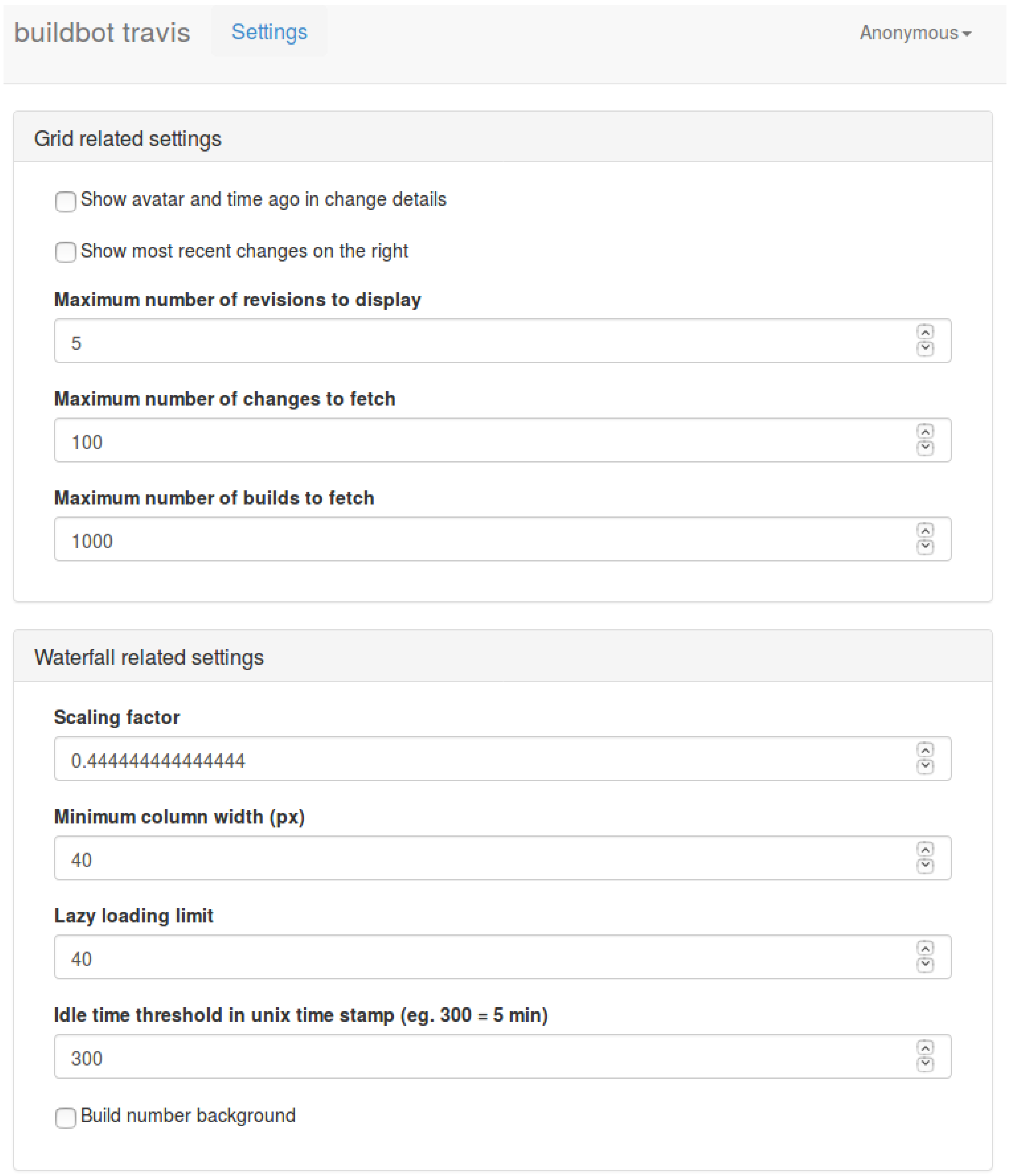
GitHub Pull Request #3845 (2 commits) (19 hours ago)

Merge pull request #3839 from Hyask/patch-1 (19 hours ago)

Figure 3.3: Sample console view

3.2.4 Settings

The settings page provides some option tweaking accessible for the user, and more important configuration options for logged in administrators. These settings can be extended in the buildbot Angular app.



Grid related settings

- [ ] Show avatar and time ago in change details
- [ ] Show most recent changes on the right

Maximum number of revisions to display: 5

Maximum number of changes to fetch: 100

Maximum number of builds to fetch: 1000

Waterfall related settings

Scaling factor: 0.4444444444444444

Minimum column width (px): 40

Lazy loading limit: 40

Idle time threshold in unix time stamp (eg. 300 = 5 min): 300

- [ ] Build number background

Figure 3.4: Sample settings menu

Chapter 4

Custom implementations

4.1 Backend componentization

By default, the Buildbot master configuration resides in the `master.cfg` file in the buildmaster's base directory. A sample file is automatically installed when a new master is created.

The config file is written in Python and it defines a dictionary named `BuildmasterConfig`, with a number of keys that are specially treated. Being a pythonic configuration, we can benefit from the features of the programming language, such as componentization of the configuration parts into modules. This comes in handy when we are dealing with large configurations. At first the `master.cfg` file seems suitable, but as we add more builders and workers, we need to keep the code readable, thus we split `master.cfg` into multiple modules.

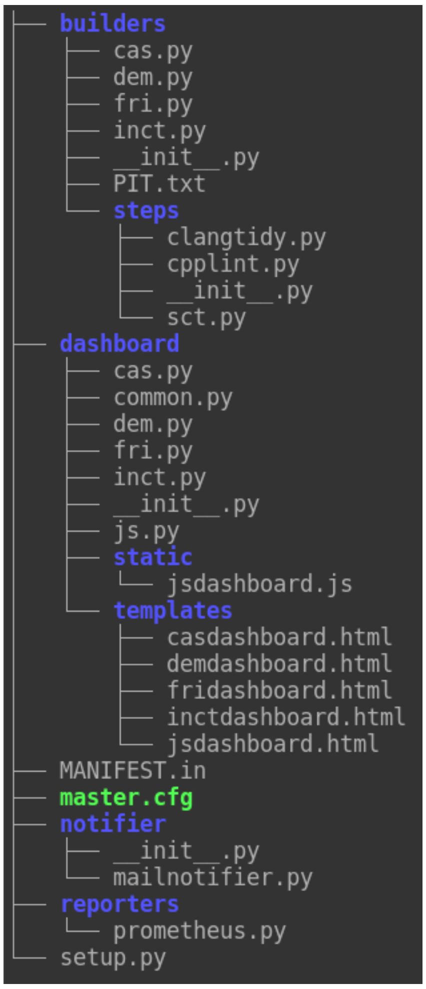
The figure below illustrates this componentization for the Nokia Buildbot project, which has 4 projects running on the same master (CAS, INCT, FRI, DEM), each of them having an arbitrary number of builders (`builders` subfolder) and a specific web dashboard (`dashboard` subfolder).

Some of the build steps have log outputs that do not match Buildbot's default parsers, so custom logparsers have been implemented (`builders/steps` subfolder).

Custom notifiers have the purpose of informing the user/admin about vital build or system information, the current implemented one uses LDAP to query the active directory in order to find the e-mail address of an user (`notifier` subfolder).

As the project increases in size, metrics become necessary. We use Prometheus to keep track of information, and Grafana to crunch it and present it in an user-friendly way (`reporters` subfolder).

All of these components could be worked into the `master.cfg` file, but the result would be an approx. 2300 line configuration file that encompasses each category, making code modifications and additions more prone to errors and most certainly a chore to go through.



```
├── builders
│   ├── cas.py
│   ├── dem.py
│   ├── fri.py
│   ├── inct.py
│   ├── __init__.py
│   ├── PIT.txt
│   └── steps
│       ├── clangtidy.py
│       ├── cpplint.py
│       ├── __init__.py
│       └── sct.py
├── dashboard
│   ├── cas.py
│   ├── common.py
│   ├── dem.py
│   ├── fri.py
│   ├── inct.py
│   ├── __init__.py
│   ├── js.py
│   ├── static
│   │   └── jsdashboard.js
│   └── templates
│       ├── casdashboard.html
│       ├── demdashboard.html
│       ├── fridashboard.html
│       ├── inctdashboard.html
│       └── jsdashboard.html
├── MANIFEST.in
├── master.cfg
├── notifier
│   ├── __init__.py
│   └── mailnotifier.py
├── reporters
│   └── prometheus.py
└── setup.py
```

Figure 4.1: Sample configuration componentization

4.1.1 Email look-up using LDAP

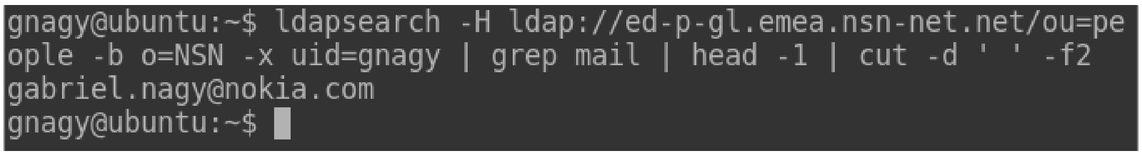
Making users aware of their build status and results is an important part of the CI process. To keep user input to a minimum, we can figure out their e-mail address from their Linux username. For this purpose, we use Python's LDAP (Lightweight Directory Access Protocol) module to search through the active directory for the user's information.

To send e-mails to users, Buildbot provides a `MailNotifier` reporter that does most of the job. This is a basic example:

```python
from buildbot.plugins import reporters
mn = reporters.MailNotifier(fromaddr="buildbot@example.org",
                            lookup="example.org")
c['services'].append(mn)
```

Listing 4.1: *mailnotifier_default.py*

Concerning our use case, we need to configure a proper `lookup` function. For builds triggered by VCS commits, this is automatically managed by adding the e-mail addresses of the committers. However, try builds do not provide the e-mail parameter, so it has to be figured out using the user's Unix username.



```
gnagy@ubuntu:~$ ldapsearch -H ldap://ed-p-gl.emea.nsn-net.net/ou=pe
ople -b o=NSN -x uid=gnagy | grep mail | head -1 | cut -d ' ' -f2
gabriel.nagy@nokia.com
gnagy@ubuntu:~$ 
```

Figure 4.2: Using `ldapsearch` to get the e-mail address from the Active Directory

For Buildbot integration, we have to implement an `IEmailLookup` interface that is provided by Buildbot. This interface has a `getAddress` method that turns an username string into a valid e-mail address.[69]

To implement the interface, we need to adapt the `ldapsearch` to Python code:

```python
import ldap
import re
from buildbot import interfaces
from buildbot import util
from twisted.python import log
from zope.interface import implementer


@implementer(interfaces.IEmailLookup)
class IcdCslToEmail(util.ComparableMixin):
    def __init__(self, emailsMap=None):
        self.emailsMap = emailsMap

    def getAddress(self, csl):
        """ It will search first in emailsMap if user is present. If not, it will
            search against ICD.
        """

        if self.emailsMap and csl in self.emailsMap is not None:
            return self.emailsMap[csl]
        l = ldap.initialize('ldap://ed-p-gl.emea.nsn-net.net')
        binddn = "o=NSN"
        basedn = "o=NSN"

        searchFilter = "uid=" + csl
        searchAttribute = ["mail"]
        searchScope = ldap.SCOPE_SUBTREE
        try:
            l.protocol_version = ldap.VERSION3
            l.simple_bind_s(binddn)
        except ldap.LDAPError as e:
            if type(e.message) == dict and 'desc' in e.message:
                log.msg(e.message['desc'])
            else:
                log.msg(e)
        try:
            ldap_result_id = l.search(basedn, searchScope, searchFilter,
                searchAttribute)
            result_set = []
            while 1:
                result_type, result_data = l.result(ldap_result_id, 0)
                if (result_data == []):
                    break
                else:
                    if result_type == ldap.RES_SEARCH_ENTRY:
                        result_set.append(result_data)
            log.msg(result_set)
            match = re.findall(r'[\w\.-]+@[\w\.-]+', str(result_set[0]))
            if match:
                return match[0]

        except ldap.LDAPError as e:
            log.msg(e)
        l.unbind_s()
```

Listing 4.2: *IEmailLookup.py*

As shown in the code, the class takes an optional argument, `emailsMap`, which is a dictionary of username and e-mail key/value pairs. This is usually done to avoid querying the Active Directory for an user that has used the system before.

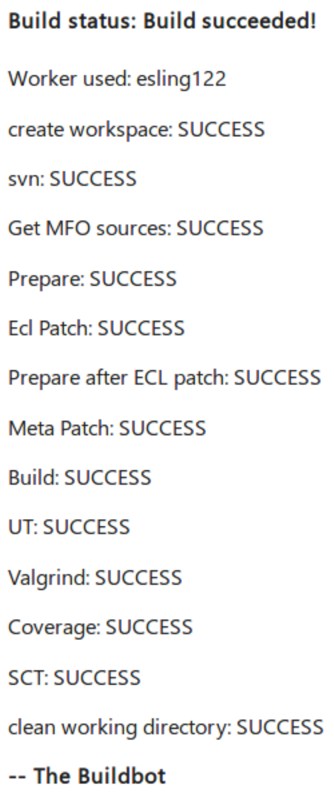
Finally, to deploy this interface, we import it in the master configuration file, and we adjust our `MailNotifier` reporter accordingly, also adding a HTML template as desired:

```
from buildbot.plugins import reporters
from mailnotifier import IcdCslToEmail

mailMap = {
    'cross_ci': 'cross_ci@alcatel-lucent.com',
}

template=u'''\
<h4>Build status: {{ summary }}</h4>
<p> Worker used: {{ workername }}</p>
{% for step in build['steps'] %}
<p> {{ step['name'] }}: {{step['state_string']}}</p>
{% endfor %}
<p><b> -- The Buildbot</b></p>
'''

m = reporters.MailNotifier(
    fromaddr="cross_ci@nokia.com", # the address from which mail is sent
    tags=["cas"], # only send for builders that have this tag; useful for
        multi-project masters
    lookup=mailnotifier.IcdCslToEmail(mailMap), # our lookup function that
        takes the mailMap dict as an argument
    buildSetSummary=True, # show a summary of the whole buildset in the e-
```

```
        mail
    messageFormatter=reporters.MessageFormatter(
        template=template, template_type='html',
        wantProperties=True, wantSteps=True)) # the HTML template to use in
            the message body

c['services'].append(m)
```

Listing 4.3: *mailnotifier_extended.py*

Now, each time user-triggered builds on builders with the `cas` tag are completed, the user receives an e-mail with the status of his build. If the user submitted a patchfile, it is also included in the e-mail.



Build status: Build succeeded!

Worker used: esling122

create workspace: SUCCESS

svn: SUCCESS

Get MFO sources: SUCCESS

Prepare: SUCCESS

Ecl Patch: SUCCESS

Prepare after ECL patch: SUCCESS

Meta Patch: SUCCESS

Build: SUCCESS

UT: SUCCESS

Valgrind: SUCCESS

Coverage: SUCCESS

SCT: SUCCESS

clean working directory: SUCCESS

-- The Buildbot

Figure 4.3: Contents of the e-mail sent by Buildbot

4.1.2 Log parsing

Buildbot provides a number of parsers for commonly used commands such as compilers (cmake, make, visual c++...) and tests (cppcheck, make test, trial...). However, there are special cases that do not fit the default parsers and custom ones have to be implemented.[68]

As it will be used in other sections, we will implement a simple parser for the `clang-tidy` command. We begin by creating a new class that inherits the default `ShellCommand`. We provide the class with a `logConsumer` function that incrementally gets every line of the command's output, and we feed the lines to the `gatherTestStatistics` function that parses the lines. We use Python's regular expressions module to search for the desired content.

```
class ClangTidy(shell.ShellCommand):
    def logConsumer(self):
        while True:
            stream, line = yield
            self.gatherTestStatistics(line)

    def gatherTestStatistics(self, line):
        m = re.search('Errors:\s*(\d+)', line)
        if m:
            self.errors = m.group(1)
        m = re.search(r'Warnings:\s*(\d+)', line)
        if m:
            self.warnings = m.group(1)
```

Listing 4.4: *clangtidy.py*

The `__init__` function initializes our new `ShellCommand`, adding a log observer to the `stdio` output, and providing initial fields for possible errors and warnings.

```
def __init__(self, **kwargs):
    shell.ShellCommand.__init__(self, **kwargs)
    self.addLogObserver('stdio', logobserver.LineConsumerLogObserver(self.
        logConsumer))
    self.errors = "0"
    self.warnings = "0"
```

Listing 4.5: *clangtidy.py*

The command is mostly done as we successfully parsed its output. However, it must be evaluated in order to set a status after the step is finished. The evaluation is straightforward, if we have something in the `errors` field, we mark the step as failed, if we have `warnings` and no `errors`, we mark the step as having warnings (the build will not stop).

```
def evaluateCommand(self, cmd):
    res = shell.ShellCommand.evaluateCommand(self, cmd)
    if self.errors != "0":
        res = FAILURE
    if self.errors == "0" and self.warnings != "0":
        res = WARNINGS
    return res
```

Listing 4.6: *clangtidy.py*

We successfully implemented our parser, however the user does not know the error/warning count until he opens the log. To fix that, we can update the step's description with the count.

```
def getResultSummary(self):
    cmdsummary = u""
    if self.errors and self.errors != "0":
        cmdsummary = "errors: " + self.errors
    if self.warnings and self.warnings != "0":
```

```python
        if cmdsummary:
            cmdsummary = cmdsummary + ", "
            cmdsummary = cmdsummary + "warnings: " + self.warnings
    return {u'step': cmdsummary}
```

Listing 4.7: *clangtidy.py*

We now have a fully working `clang-tidy` command parser. It is a good practice to write unit tests for our code, so we make a generic test suite:

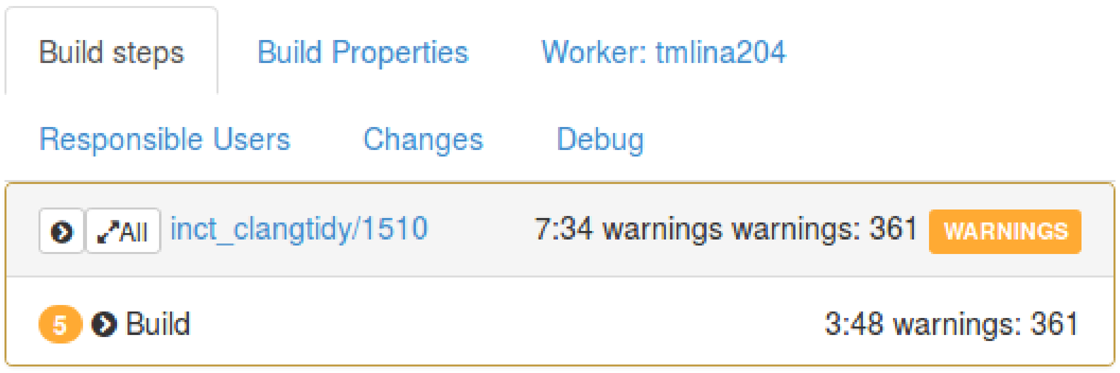
```python
import unittest

class TestSummaryEvaluation(unittest.TestCase):
    def setUp(self):
        pass

    def test_error_tests_line(self):
        line = "Errors: 1690"
        bs = ClangTidy()
        bs.gatherTestStatistics(line)
        self.assertEqual("1690", bs.errors)

    def test_warning_tests_line(self):
        line = "Warnings: 166"
        bs = ClangTidy()
        bs.gatherTestStatistics(line)
        self.assertEqual("166", bs.warnings)

    def test_all_in_block_passed(self):
        line = """=====================
               ClangTidy summary
             =====================
             Errors: 0
             Warnings: 166"""
        bs = ClangTidy()
        bs.gatherTestStatistics(line)
        self.assertEqual("0", bs.errors)

    def test_all_in_block_failed(self):
        line = """=====================
               ClangTidy summary
             =====================
             Errors: 614
             Warnings: 154"""
        bs = ClangTidy()
        bs.gatherTestStatistics(line)
        self.assertEqual("614", bs.errors)
        self.assertEqual("154", bs.warnings)
```

```
if __name__ == "__main__":
    unittest.main()
```

Listing 4.8: *clangtidy_test.py*

All that remains to do is to import this class into the configuration file, and create the step using `steps.ClangTidy` instead of `steps.ShellCommand`.

Buildbot now correctly processes the output and updates the result accordingly:



```
inct_clangtidy/1510    7:34 warnings warnings: 361  WARNINGS
5  Build               3:48 warnings: 361
```

Figure 4.4: clang-tidy command output

4.1.3 Preferential build steps

In our use cases, there is the possibility of patching more than one component in a build. By default, Buildbot executes all commands until failed.

`BuildSteps` are usually specified in the buildmaster's configuration file, in a list that goes into the `BuildFactory`. The `BuildStep` instances in this list are used as templates to construct new independent copies for each build (so that state can be kept on the `BuildStep` in one build without affecting a later build). Each `BuildFactory` can be created with a list of steps, or the factory can be created empty and then steps added to it using the `addStep` method:

```
from buildbot.plugins import util, steps

f = util.BuildFactory()
f.addSteps([
    steps.SVN(repourl="http://svn.example.org/trunk/"),
    steps.ShellCommand(command=["make", "all"]),
    steps.ShellCommand(command=["make", "test"])
])
```

Listing 4.9: *simple_factory.py*

However, we would like to execute only certain commands, depending on the status of the previous commands. To accomplish that, we need to know that Buildbot steps finish with a status described by one of four values defined in `buildbot.status.builder`: `SUCCESS`, `WARNINGS`, `FAILURE` or `SKIPPED`.[67]

Let's say that we have a builder that checks out a source, runs a linter and builds the code only if the lint step had no errors, then cleans up the build folder. An example for this use case can be seen below:

```
from buildbot.plugins import util, steps
from buildbot.process.results import FAILURE

f = util.BuildFactory()
f.addSteps([
    steps.SVN(repourl="http://svn.example.org/trunk/"),
    steps.ClangTidy(command=["clang-tidy", "src/*", "--", "-std=c++11"]),
    steps.Compile(command=["make", "test"], doStepIf=lambda step: step.build.
        executedSteps[-2].results != FAILURE),
    steps.ShellCommand(command=["rm", "-rf", "build/*"], alwaysRun=True)
])
```

Listing 4.10: *preferential_factory.py*

To skip a step, Buildbot provides us with the `doStepIf` parameter for a `ShellCommand` . The value is set to a lambda function that returns `True` only if the previous step's result is `FAILURE`.

`step.build.executedSteps` is a list of the executed steps including the one that is running when the list is queried. We get the result of the `clang-tidy` command by selecting the second to last executed step. In Python this is easily done by going backwards on the list, providing it with a negative value (`[-2]`).

At the end, we want to clean the workspace so it does not interfere with future builds. By default, Buildbot stops a build when a step fails. To override that, we provide the `rm -rf` command with the `alwaysRun` parameter.

4.2 Web interface development

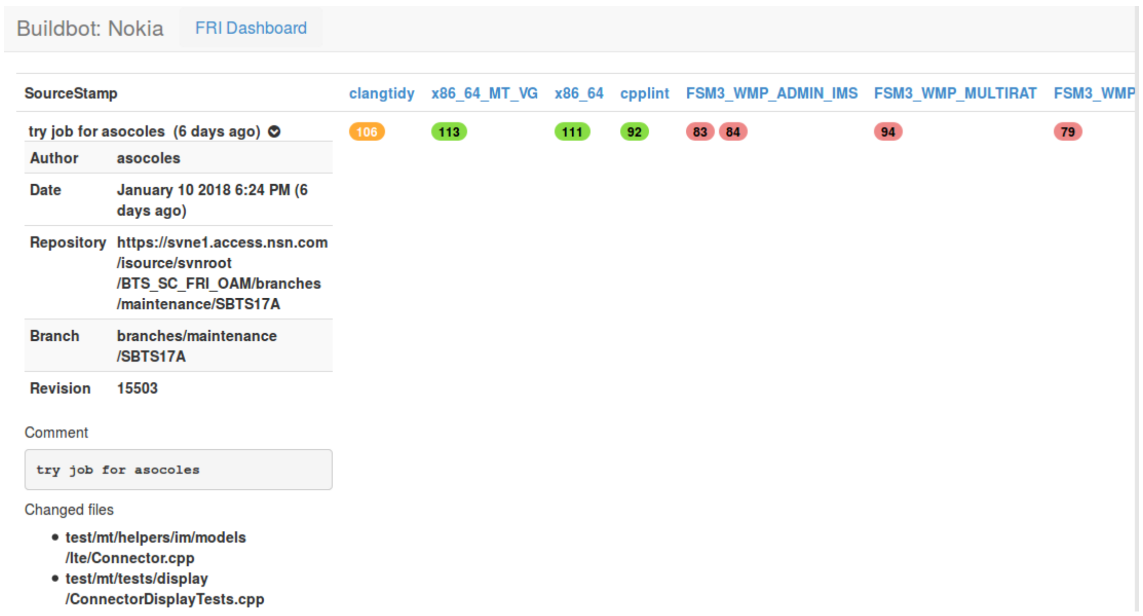
While Buildbot's default web views (console, grid, waterfall) may be useful for some users, the real advantage lies in their customizability. We will build the same custom dashboard twice, once with Flask/WSGI, and once with Angular. At the end, we will compare and contrast the two approaches.

4.2.1 Flask/WSGI dashboards

The simpler way, for most Python developers, is to use the `buildbot_wsgi_dashboards` plugin to write a server side generated dashboard and integrate it in the UI.

The plugin is compatible with any WSGI web framework, we will use Flask as it is one of the more popular ones. A `/index.html` route needs to be implemented, which will render the code representing the dashboard. The application framework runs in a thread outside of Twisted and it provides a synchronous wrapper for accessing the Data API. The HTML output is ran inside the Angular application, using its CSS and some of the directives defined by the Buildbot UI. It also has full access to the application JS context.[71]

For our custom dashboard, we need to be able to filter by tag, and to show both poller-triggered builds and manually triggered builds. Our end result will look similar to this figure:


Buildbot: Nokia FRI Dashboard

SourceStamp	clangtidy	x86_64_MT_VG	x86_64	cpplint	FSM3_WMP_ADMIN_IMS	FSM3_WMP_MULTIRAT	FSM3_WMP
try job for asocoles (6 days ago)	106	113	111	92	83 84	94	79

Author	asocoles
Date	January 10 2018 6:24 PM (6 days ago)
Repository	https://svne1.access.nsn.com/isource/svnroot/BTS_SC_FRI_OAM/branches/maintenance/SBTS17A
Branch	branches/maintenance/SBTS17A
Revision	15503

Comment

```
try job for asocoles
```

Changed files

- test/mt/helpers/im/models/lte/Connector.cpp
- test/mt/tests/display/ConnectorDisplayTests.cpp

Figure 4.5: Custom dashboard result

We begin by creating the logic for the dashboard. First we create the helper functions for obtaining the changed file list and the fake change creator, since Buildbot doesn't generate one when triggering a manual build:

```python
from buildbot.util import datetime2epoch


def get_changed_files(patch_body):
    file_list = []
    files = patch_body.split("Index: ")

    for f in files:
        f2 = f.split("=")
        f2[0] = f2[0].strip(" :")
        file_list.append(f2[0])

    return file_list[1:]


def getChangeFromBbSourceStamp(sourcestampdetails):
    fakeChange = {
        "revision": str(sourcestampdetails['revision']),
        "when_timestamp": datetime2epoch(sourcestampdetails['created_at']),
        "codebase": str(sourcestampdetails['codebase']),
        "repository": str(sourcestampdetails['repository']),
        "branch": str(sourcestampdetails['branch']),
    }
    if sourcestampdetails['patch']:
        fakeChange["files"] = get_changed_files(sourcestampdetails['patch']['body'])
        fakeChange["author"] = str(sourcestampdetails['patch']['author'])
        fakeChange["comments"] = str(sourcestampdetails['patch']['comment'])
    return fakeChange
```

Listing 4.11: *helper_functions.py*

We will use of this data when filling in the template fields. By making use of Buildbot's Data API we obtain information about the builders and buildsets that we want to display:

```
import os
import time
from flask import Flask
from buildbot.data.resultspec import Filter

fridashboardapp = Flask('test', root_path=os.path.dirname(__file__))
fridashboardapp.config['TEMPLATES_AUTO_RELOAD'] = True


@fridashboardapp.route("/index.html")
def main():
    bigData = {}
    epoch_time = int(time.time())
    earliestSubmittedTime = epoch_time - (60 * 60 * 24 * 7) # last week
    out_file = open("/tmp/buildotTmpLogFriDash", "w")
    fribuilders = fridashboardapp.buildbot_api.dataGet("/builders", filters=[Filter(
        "tags", "contains", ["fri"]), Filter("masterids", "ne", [[]])])
    buildsets = fridashboardapp.buildbot_api.dataGet("buildsets", limit=128, order=[
        "-bsid"], filters=[Filter("submitted_at", "gt", [earliestSubmittedTime])])
```

Listing 4.12: *flask_dashboard.py*

Next, we iterate through the buildsets, keeping the ones that interest us, and saving the information that we want to show in the template:

```
    for buildset in buildsets:
        sourcestamps = buildset['sourcestamps']
        if len(sourcestamps[0]) != 1:
            out_file.write("ERROR: more that 1 sourcestamp for buildset" + str(
                buildset["bsid"]) + "\n")
        sourcestamp = sourcestamps[0]
        proj = sourcestamp['project']
        if proj != "fri": # only go on if we have the desired project
            continue

        ssid = sourcestamp['ssid']
        if ssid not in bigData:
            bigData[ssid] = {}
            bigData[ssid]['change'] = getChangeFromBbSourceStamp(sourcestamp)
            bigData[ssid]['builders'] = {}

        buildrequests = fridashboardapp.buildbot_api.dataGet("buildrequests",
            filters=[Filter("buildsetid", "eq", [buildset['bsid']])])
        for buildrequest in buildrequests:
            id = buildrequest['builderid']
            if id not in bigData[ssid]['builders']:
                bigData[ssid]['builders'][id] = []

            if buildrequest["claimed"]:
                builds = fridashboardapp.buildbot_api.dataGet(("buildrequests",
                    buildrequest["buildrequestid"], "builds"))
                for build in builds:
                    results_text = statusToString(build['results']).upper()
                    if results_text == 'NOT FINISHED':
                        results_text = 'PENDING pulse'

                    bigData[ssid]['builders'][id].append({
                        'type': 'build',
                        'number': build["number"],
                        'results_text': results_text
                    })
            else:
                bigData[ssid]['builders'][id].append({
                    'type': 'buildrequest',
                    'id': buildrequest["buildrequestid"],
                    'results_text': "UNKNOWN"
                })
```

Listing 4.13: *flask_dashboard.py*

At the end we return the template with our parsed content, the builders and a dictionary of all the related data we want to show:

```
return render_template('fridashboard.html', builders=fribuilders, bigdata=bigData)
```

Listing 4.14: *flask_dashboard.py*

To keep the page small, we remove the bloat by minifying our HTML response. This can be easily done in Flask with the help of the `@after_request` decorator:

```
@fridashboardapp.after_request
def response_minify(response):
    from htmlmin.main import minify
    if response.content_type == u'text/html; charset=utf-8':
        response.set_data(
            minify(response.get_data(as_text=True))
        )
        return response
    return response
```

Listing 4.15: *flask_dashboard.py*

For the HTML part, we generate a simple table, filling it out with the data returned by our Flask file

```
<div class="container mydashboard">
```

```
    <!-- Create a table of builds organised by builders in columns -->
    <table class="table">
      <tr><th>SourceStamp</th>
        {% for builder in builders %}
        <th><a ng-href="#/builders/{{builder.builderid}}" class="ng-
            binding" href="#/builders/{{builder.builderid}}">{{builder.
            name}}</a></th>
        {% endfor %}
      </tr>
      {% for ssid, data in bigdata.items() | sort(reverse=True) %}
      <tr>
        <th><changedetails change="{{data.change}}"/></th>
        {% for builder in builders %}
        <th>
          {% for build in data.builders[builder.builderid] | sort() %}
          {% if build.type == "build" %}
          <a class="badge-status badge results_{{build.results_text}}"
             href="#/builders/{{builder.builderid}}/builds/{{build.number
             }}">{{build.number}}</a>
          {% endif %}
          {% if build.type == "buildrequest" %}
          <a class="badge-status badge results_{{build.results_text}}"
             href="#/buildrequests/{{build.id}}">{{build.id}}</a>
          {% endif %}
          {% endfor %}
        </th>
        {% endfor %}
      </tr>
      {% endfor %}
      </tr>
    </table>
  </div>
```

Listing 4.16: *flask_template.html*

The only thing left to do is to load our dashboard in the `master.cfg` file. We do this by importing our Flask file and appending to the `c['www']['plugins']['wsgi_dashboards']` key:

```
import flask_template

c['www']['plugins']['wsgi_dashboards'].append(
    {
        'name': 'fridashboard', # as used in URLs
        'caption': 'FRI Dashboard', # Title displayed in the UI'
        'app': flask_template.fridashboardapp,
        # priority of the dashboard in the left menu (lower is higher
            in the
        # menu)
        'order': 5,
        # available icon list can be found at http://fontawesome.io/
            icons/
        'icon': 'bars'
    }
)
```

Listing 4.17: *master.cfg*

After we restart/reconfig our Buildbot, our new dashboard can be seen in the sidebar:

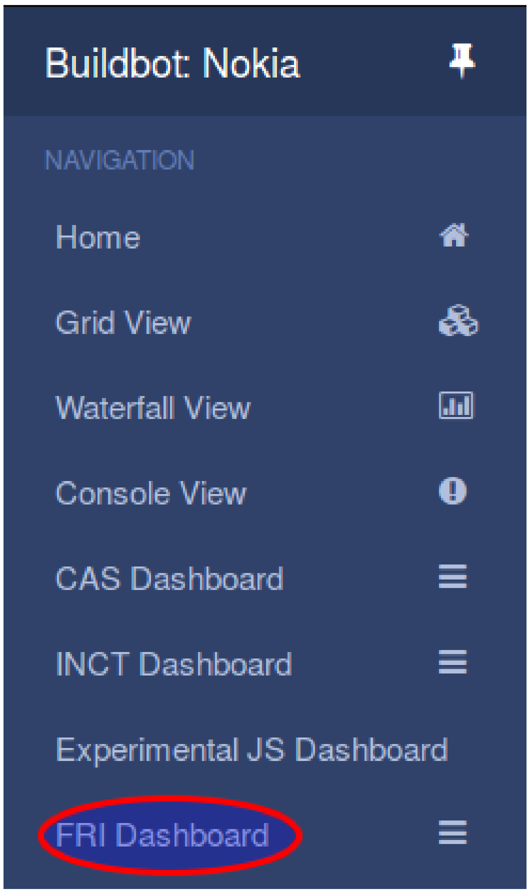


Figure 4.6: Newly modified sidebar

4.2.2 Angular dashboards

Though easy to implement and familiar to Python developers, the Flask dashboard is not efficient for long term use, since it is not asynchronously updated when builds are added, and the whole page has to be reloaded, making this a very time and resource consuming activity.

The right solution would be to follow the approach of *Console* and *Waterfall view* and create these dashboards as Buildbot plugins using Angular. By choosing this approach, we benefit from real-time loading with AJAX, reducing our load by a large margin. For developers unfamiliar with Node.js and its practices, this approach may seem unfriendly. However, the developers have put up a "quick-start guide" for Angular live coding.[72] In short, one has to locally build the frontend of Buildbot, scaffold a dashboard template and install a browser plugin in order to benefit from live reloading when the code changes.

Using CoffeeScript and Jade, we recreate the dashboard logic previously used with Flask.

4.3 User scripts

user try scripts to send patches with uncomitted code to buildbot for testing

4.4 Capturing metrics

buildbot metrics + Prometheus \w alertmanager + grafana

4.5 Extending the source code

extending buildbot source to allow multiple patchfiles and more API entrypoints maybe?

Chapter 5

Conclusion

Bibliography

[1] McConnell, Steve, Code Complete, Microsoft Press, 2009, p. 100 2.1.1

[2] About Python, URL: https://www.python.org/about/ 2.1.1

[3] TIOBE Index for November 2017, URL: https://www.tiobe.com/tiobe-index/ 2.1.1

[4] Prechelt, Lutz, An empirical comparison of C, C++, Java, Perl, Python, Rexx, and TCL, Fakultät für Informatik, Universität Karlsruhe, 2000 2.1.1

[5] McKellar, J. and Fettig, A., Twisted Network Programming Essentials, O'Reilly Media, 2013, p. xiii 2.1.2

[6] McKellar, J. and Fettig, A., Twisted Network Programming Essentials, O'Reilly Media, 2013, p. 11 2.1.2.4

[7] Twisted Framework: Core ideas, URL: https://en.wikipedia.org/w/index.php?title=Twisted_(software)&oldid=808304626 2.1.2.3

[8] Owens, Michael, The Definitive Guide to SQLite, Apress, 2014, p. 133 2.1.3

[9] Most Widely Deployed SQL Database Engine, URL: https://sqlite.org/mostdeployed.html 2.1.3

[10] Introduction - Buildbot latest documentation, URL: http://docs.buildbot.net/current/manual/introduction.html 2.1

[11] Database - Buildbot latest documentation, URL: http://docs.buildbot.net/latest/developer/database.html 2.1.3

[12] Global Configuration - Buildbot latest documentation, URL: http://docs.buildbot.net/latest/manual/cfg-global.html 2.1.3

[13] Angular - Architecture Overview, URL: https://angular.io/guide/architecture 2.2.2

[14] AngularJS - Wikipedia, URL: https://en.wikipedia.org/w/index.php?title=AngularJS&oldid=812169171 2.2.2

[15] AngularJS: Developer Guide: Introduction, URL: https://docs.angularjs.org/guide/introduction 2.2.2

[16] What is Node.js? - Definition from WhatIs.com, URL: http://whatis.techtarget.com/definition/Nodejs 2.2.3

[17] A Beginner's Guide to npm - the Node Package Manager, URL: https://www.sitepoint.com/beginners-guide-node-package-manager 2.2.4

[18] Ampersand.js - Learn, URL: https://ampersandjs.com/learn/npm-browserify-and-modules 2.2.4

[19] Ojamaa, Andres; Duuna, Karl, Assessing the Security of Node.js Platform, URL: http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=6470829 2.2.4

[20] Understanding npm, URL: https://unpm.nodesource.com 2.2.4

[21] npm Code of Conduct: acceptable package content, URL: https://www.npmjs.com/policies/conduct#acceptable-package-content 2.2.4

[22] npm-stat: download statistics for NPM packages, URL: https://npm-stat.com 2.2.4

[23] How To Use npm to Manage Node.js Packages on a Linux Server, URL: https://www.digitalocean.com/community/tutorials/how-to-use-npm-to-manage-node-js-packages-on-a-linux-server 2.2.4

[24] npm-install, URL: https://docs.npmjs.com/cli/install 2.2.4, 2.2.5

[25] semver, URL: https://docs.npmjs.com/misc/semver 2.2.4

[26] npm-version, URL: https://docs.npmjs.com/cli/version 2.2.4

[27] The npm Blog - Hello, Yarn!, URL: http://blog.npmjs.org/post/151660845210/hello-yarn 2.2.4

[28] Yarn: A new package manager for JavaScript | Engineering Blog | Facebook Code, URL: https://code.facebook.com/posts/1840075619545360/yarn-a-new-package-manager-for-javascript 2.2.4

[29] Mao, Jed; Schmitt, Maximilian; Stryjewski, Tomasz; Country Holt, Cary; Lubelski, Wiliam, Developing a Gulp Edge (1st ed.), Bleeding Edge Press, 2014 2.2.5

[30] substack/stream-handbook: how to write note programs with streams, URL: https://github.com/substack/stream-handbook 2.2.5

[31] gulpjs/gulp: Writing a plugin, URL: https://github.com/gulpjs/gulp/blob/master/docs/writing-a-plugin/README.md 2.2.5

[32] Building With Gulp - Smashing Magazine, URL: https://www.smashingmagazine.com/2014/06/building-with-gulp 2.2.5

[33] gulpjs/gulp: gulp CLI docs, URL: https://github.com/gulpjs/gulp/blob/master/docs/CLI.md 2.2.5

[34] Gulp for Beginners, URL: https://css-tricks.com/gulp-for-beginners 2.2.5

[35] gulpjs/gulp: Getting Started, URL: https://github.com/gulpjs/gulp/blob/master/docs/getting-started.md 2.2.5

[36] Maynard, Travis, Getting Started with Gulp, Packt Publishing Ltd., 2015 2.2.5

[37] MacCaw, Alex, The Little Book on CoffeeScript (1st ed.), O'Reilly Media, 2012 2.2.6

[38] CoffeeScript Cookbook - String Interpolation, URL: https://coffeescript-cookbook.github.io/chapters/strings/interpolation 2.2.6.1

[39] CoffeeScript Cookbook - Comparing Ranges, URL: https://coffeescript-cookbook.github.io/chapters/syntax/comparing_ranges 2.2.6.1

[40] CoffeeScript Cookbook - Embedding JavaScript, URL: https://coffeescript-cookbook.github.io/chapters/syntax/embedding_javascript 2.2.6.1

[41] coffeescript -> javascript ES6 (ES2015) ? · Issue \#3804 · buildbot/buildbot - Embedding JavaScript, URL: https://github.com/buildbot/buildbot/issues/3804 2.2.6

[42] CoffeeScript Coding Style - Buildbot latest documentation, URL: http://docs.buildbot.net/latest/developer/coffeescript-style.html 2.2.6

[43] Getting started with Pug template engine - Codeburst, URL: https://codeburst.io/getting-started-with-pug-template-engine-e49cfa291e33 2.2.7

[44] Concepts - Buildbot latest documentation: Source Stamps, URL: http://docs.buildbot.net/latest/manual/concepts.html#source-stamps 3.1.1

[45] Concepts - Buildbot latest documentation: Who, URL: http://docs.buildbot.net/latest/manual/concepts.html#who 3.1.2.1

[46] Concepts - Buildbot latest documentation: Files, URL: http://docs.buildbot.net/latest/manual/concepts.html#files 3.1.2.2

[47] Concepts - Buildbot latest documentation: Comments, URL: http://docs.buildbot.net/latest/manual/concepts.html#comments 3.1.2.3

[48] Concepts - Buildbot latest documentation: Project, URL: http://docs.buildbot.net/latest/manual/concepts.html#project 3.1.2.4

[49] Concepts - Buildbot latest documentation: Repository, URL: http://docs.buildbot.net/latest/manual/concepts.html#repository 3.1.2.5

[50] Concepts - Buildbot latest documentation: Codebase, URL: http://docs.buildbot.net/latest/manual/concepts.html#codebase 3.1.2.6

[51] Concepts - Buildbot latest documentation: Revision, URL: http://docs.buildbot.net/latest/manual/concepts.html#revision 3.1.2.7

[52] Concepts - Buildbot latest documentation: Change Properties, URL: http://docs.buildbot.net/latest/manual/concepts.html#change-properties 3.1.2.9

[53] Concepts - Buildbot latest documentation: Branches, URL: http://docs.buildbot.net/latest/manual/concepts.html#branches 3.1.2.8

[54] Scheduling Builds - Buildbot latest documentation, URL: http://docs.buildbot.net/latest/manual/concepts.html#scheduling-builds 3.1.3

[55] Buildsets - Buildbot latest documentation, URL: http://docs.buildbot.net/latest/manual/concepts.html#buildsets 3.1.4

[56] BuildRequests - Buildbot latest documentation, URL: http://docs.buildbot.net/latest/manual/concepts.html#buildrequests 3.1.5

[57] Builders - Buildbot latest documentation, URL: http://docs.buildbot.net/latest/manual/concepts.html#builders 3.1.6

[58] Workers - Buildbot latest documentation, URL: http://docs.buildbot.net/latest/manual/concepts.html#workers 3.1.8

[59] Builds - Buildbot latest documentation, URL: http://docs.buildbot.net/latest/manual/concepts.html#builds 3.1.7

[60] Users - Buildbot latest documentation, URL: http://docs.buildbot.net/latest/manual/concepts.html#users 3.1.9

[61] Build Properties - Buildbot latest documentation, URL: http://docs.buildbot.net/latest/manual/concepts.html#build-properties 3.1.10

[62] WWW Server - Buildbot latest documentation, URL: http://docs.buildbot.net/latest/developer/www-server.html 3.2

[63] Data API - Buildbot latest documentation, URL: http://docs.buildbot.net/latest/developer/data.html#data-api 3.1.11.3

[64] Grid View - Buildbot latest documentation, URL: http://docs.buildbot.net/latest/manual/cfg-www.html#grid-view 3.2.1

[65] Console View - Buildbot latest documentation, URL: http://docs.buildbot.net/latest/manual/cfg-www.html#console-view 3.2.3

[66] Waterfall View - Buildbot latest documentation, URL: http://docs.buildbot.net/latest/manual/cfg-www.html#waterfall-view 3.2.2

[67] Build Steps - Buildbot latest documentation, URL: http://docs.buildbot.net/latest/manual/cfg-buildsteps.html 4.1.3

[68] Customization - Buildbot latest documentation: Adding Log Observers, URL: http://docs.buildbot.net/latest/manual/customization.html#adding-logobservers 4.1.2

[69] buildbot.interfaces.IEmailLookup, URL: http://docs.buildbot.net/0.8.3/reference/buildbot.interfaces.IEmailLookup-class.html 4.1.1

[70] Foreword - Flask Documentation (0.12), URL: http://flask.pocoo.org/docs/0.12/foreword/ 2.2.1

[71] Customization - Buildbot latest documentation: Writing dashboards with Flask or Bottle, URL: http://docs.buildbot.net/latest/manual/customization.html#writing-dashboards-with-flask-or-bottle 4.2.1

[72] Buildbot UI Plugin for Python developer - Buildbot - Medium, URL: https://medium.com/buildbot/buildbot-ui-plugin-for-python-developer-ef9dcfdedac0 4.2.2

[73] Grid View - Buildbot latest documentation: Source Stamps, URL: http://docs.buildbot.net/latest/manual/cfg-www.html#grid-view

[74] Grid View - Buildbot latest documentation: Source Stamps, URL: http://docs.buildbot.net/latest/manual/cfg-www.html#grid-view

[75] Grid View - Buildbot latest documentation: Source Stamps, URL: http://docs.buildbot.net/latest/manual/cfg-www.html#grid-view

[76] Grid View - Buildbot latest documentation: Source Stamps, URL: http://docs.buildbot.net/latest/manual/cfg-www.html#grid-view

[77] Grid View - Buildbot latest documentation: Source Stamps, URL: http://docs.buildbot.net/latest/manual/cfg-www.html#grid-view