DD2480 Software Engineering Fundamentals

Lecture 3 Part 1: Continuous integration

Outline

- What is continuous integration?
- Travis Continuous Integration Service
    - Webhooks mechanism

What is continuous integration?

Definition:

- *Continuous Integration (CI) is a software development practice where members of a team integrate their work frequently, usually each person integrates at least daily - leading to multiple integrations per day.*

- *Each integration is verified by an automated build (including test) to detect integration errors as quickly as possible.*

- *Main motivation for CI: reduce integration problems and speed up a development of cohesive software.*

Problem of integration

- *Integration Hell* -- time in production when members on a delivery team integrate their individual code.
  - In traditional software development environments, hours or days of fixing the code so that it can finally integrate

Problem of integration

- *Integration Hell* -- time in production when members on a delivery team integrate their individual code.
    - In traditional software development environments, hours or days of fixing the code so that it can finally integrate
- CI: treat integration as a non-event.

- Any individual developer's work is only a few hours away from a shared project state and can be integrated back into that state in minutes.
- Any integration errors are found rapidly and can be fixed rapidly.

- The term 'Continuous Integration' originated with Kent Beck's *Extreme Programming* development process (one of its original practices)

CI: workflow. Step1: Checking out

- Begin by taking a copy of the current integrated source onto local development machine.
  - by checking out a working copy from the mainline.
- A source code control system (e.g., GitHub) keeps all of a project's source code in a repository.
- The current state of the system called the 'mainline'.
- At any time a developer can make a controlled copy of the mainline onto their own machine, i.e., 'check out'.
- The copy on the developer's machine is called a 'working copy'.

CI: workflow. Step2: Modifying the working copy and creating the local build

- Working copy can be manipulated in different ways depending on the task:
- altering the production code or adding or changing automated tests (often using a version of XUnit testing frameworks)
- Once done: carry out an automated build on own development machine.
- This takes the source code in the working copy, compiles and links it into an executable, and runs the automated tests.
- Only if it all builds and tests without errors, the overall build is considered to be good.

CI: workflow. Step3: Committing to the repository

If a good build on the working copy is obtained

- Plan committing the changes into the repository.
  - Meanwhile other people made changes to the mainline.
- First update own working copy with their changes and rebuild.
  - Others changes clash with own changes result in failure either in the compilation or in the tests.
  - Own responsibility to fix and repeat until a working copy that is properly synchronized with the mainline is built
- Once done, the developer can finally commit changes into the mainline, which then updates the repository.

CI workflow: Step 4: build on integration machine

After commit to the mainline

- Build again on an integration machine based on the mainline code.
- Only when this build succeeds the changes are done.
- The integration build can be executed manually by the developer or done automatically by the CI server

- A clash occurs between two developers: caught when the second developer to commit builds their updated working copy.
  - The integration build fails.
- The error is detected rapidly and should be quickly fixed to get the build working properly again
- In CI a failed integration build should not stay failed for long.
- A good team should have many correct builds a day.

The overall goal of CI

- Always have a stable piece of software that works properly and contains only a few bugs.
- Everybody develops off that shared stable base and never gets too far away from that base
- Achieved by the main practices of CI

Practices of CI: Maintain a Single Source Repository

- Software projects have many files that need to be orchestrated together to build a product.
  - Keeping track of all of them is a major effort

- Source Code Management Tools - configuration management, version control systems, repositories etc

- Main rule: everything should be in the repository.
  - including: test scripts, properties files, database schema, install scripts, third party libraries etc.

Practices of CI: Maintain a Single Source Repository

- The basic rule of thumb:
- One should be able to walk up to the project with a virgin machine, do a checkout, and be able to fully build the system.
  - Only a minimal amount of things should be on the virgin machine – e.g. operating system, Java development environment, or base database system
- Branches: “Nothing in excess except moderation”
- Version control systems allow us to create multiple branches, i.e., to handle different streams of development.
- It's frequently overused. Keep the use of branches to a minimum and always maintain a clear **mainline**:
  - Reasonable branches are bug fixes of prior production releases and temporary experiments

Practices of CI: Automate the Build

- Build: getting the sources turned into a running system
  - Involves compilation, moving files around, loading schemas into the databases etc
- Automated environments for builds are common
  - Unix make
  - Java Ant, Gradle, Maven
  - the .NET community -- Nant and MSBuild.
- Make sure you can build and launch your system using these scripts using a single command.

Practices of CI: Automate the Build

- A big build often takes long time
  - Irritating if only a small change is made.
- A good build tool analyzes what needs to be changed as part of the process.
- The common way to do this is to check the dates of the source and object files and only compile if the source date is later.
  - Dependencies are tricky: in case of dependencies between the objects they should also be rebuilt.
  - Compilers may or may not handle this
- What actually needs to be built: a system with or without test code, or with different sets of tests.
- A script should allow you to build alternative targets for different cases.

Practices of CI: Make Your Build Self-Testing

- A program may run, but do not do the right thing – correctness?
- Include automated tests in the build process.
  - Testing isn't perfect but it can catch a lot of bugs

- Extreme Programming (XP) and Test Driven Development (TDD) popularize *self-testing* code
- Self-testing code has a suite of automated tests that can check a large part of the code base for bugs.
  - The tests are self-checking and kicked off from a simple command
  - The result of running the test suite should indicate if any test failed.

- For a build to be self-testing, the failure of a test should cause the build to fail.

Practices of CI: Make Your Build Self-Testing

- TDD has popularized the XUnit family of open-source
- Tests don't prove the absence of bugs.
- With a self-testing build you might not get perfect tests.
- But imperfect tests, run frequently, are better than perfect tests never written and run
- “Broken window syndrome”

Broken window theory

Practices of CI: Everyone Commits To the Mainline (at least) Every Day

- Integration is primarily about communication.
  - For the developer to commit to the mainline: they can correctly build their code.
- Long time spent without commit – more potential conflicts
  - Conflicts that stay undetected for weeks can be very hard to resolve.
- With frequent commits: *diff-debugging* to help to resolve the conflict.
- Rule of thumb: every developer should commit to the repository every day.

Practices of CI: Every Commit Should Build the Mainline on an Integration Machine

- Using daily commits, a team gets frequent tested builds.
    - The mainline stays in a healthy state.
- Problems in practice:
    - discipline: people not doing an update and build before they commit.
    - environmental differences between developers' machines.
- Ensure that regular builds happen on an integration machine and only if this integration build succeeds should the commit be considered to be done.
- How: using a manual build or a continuous integration server.
- Monitor the progress of started integration build: only if build succeeded, you are done with your commit.

The role of CI server

- It acts as a monitor to the repository.
- Every time a commit against the repository finishes, the server automatically checks out the sources onto the integration machine, initiates a build, and notifies the committer of the result of the build.
- The committer isn't done until she gets the notification - usually an email.

Practices of CI: Fix Broken Builds Immediately

- If the mainline build fails, it needs to be fixed right away.
- The team should always be developing on a known stable base.
- Often the fastest way to fix the build is to revert the latest commit from the mainline.
- Backward recovery: taking the system back to the last-known good build.
  - Unless the cause for the breakage is immediately obvious, just revert the mainline and debug the problem on a development workstation.

Practices of CI: Keep the Build Fast

- CI should provide rapid feedback but a build can take a long time.
- XP guideline: a ten minute build
  - Most of our modern projects achieve this.
- Worth investing efforts: frequent commits add up to a lot of time.

Practices of CI: Deployment pipeline

- **Deployment pipeline** (also known as **build pipeline** or **staged build**) multiple builds done in sequence.
- The **commit build** is the build that's needed when someone commits to the mainline.
  - Should be done quickly. Has shortcuts and reduced ability to detect bugs.

- A two stage deployment pipeline:
- The first stage -- compilation and run tests that are more localized unit tests with the database completely stubbed out.
  - Fast tests but does not find bugs involving larger scale interactions, e.g., real database
- The second stage build runs a different suite of tests that do hit the real database and involve more end-to-end behavior.
  - Might take a couple of hours to run.
- The second-stage build runs when it can, picking up the executable from the latest good commit build for further testing.
- Ensure that any later-stage failure leads to new tests in the commit build that would have caught the bug
  - Strengthen the commit tests

Practices of CI: Make it Easy for Anyone to Get the Latest Executable and Ensure Visibility

- Any team member should be able to get the latest executable and be able to run it: for demonstrations, exploratory testing, etc.
- Remember: work for your customer and build the right software. Typically hard to say what is wanted in advance: easier to comment on what needs to be changed.
- Everyone knows where to find the latest executable.
- Visibility: communicate is the state of the mainline build.

Benefits of Continuous Integration

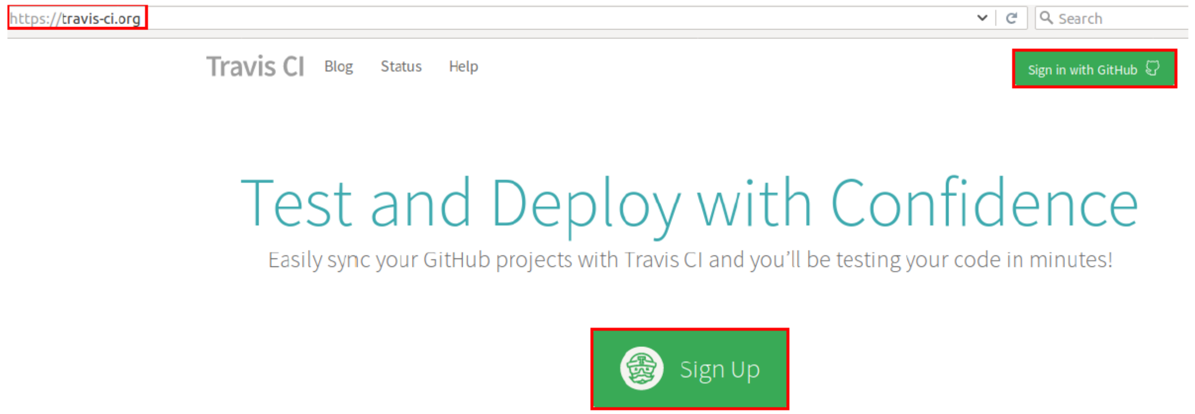
- Reduced risk.
  - Deferred integration: hard to predict how long it will take to do, and what is the progress.
- Bugs in deployed software: lost customers
- CI doesn't get rid of bugs, but it does make them dramatically easier to find and remove.
- Self-testing code.
  - If you introduce a bug and detect it quickly it's far easier to get rid of.
  - It's fresh in your memory
- Bugs are cumulative. The more bugs you have, the harder it is to remove each one

Travis CI

- https://travis-ci.org/
- Easy to use build server for projects hosted on GitHub
- Projects can be tested and deployed.
- Pull requests (PRs) can also be built automatically with Travis CI.

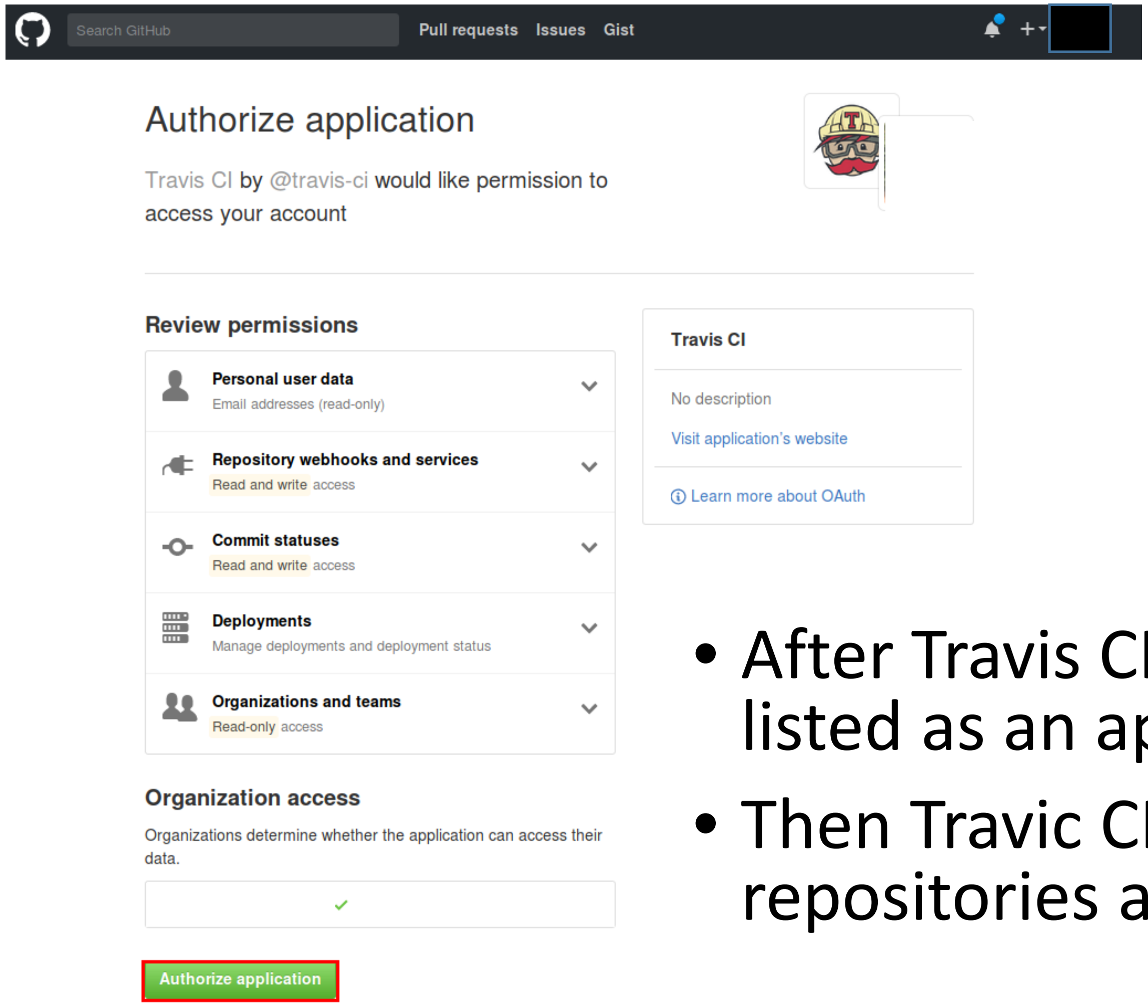
Setting up

- Go to https://travis-ci.org/ and press the Sign Up or Sign in with GitHub button.


https://travis-ci.org
Search
Travis CI
Blog
Status
Help
Sign in with GitHub
Test and Deploy with Confidence
Easily sync your GitHub projects with Travis CI and you'll be testing your code in minutes!
Sign Up

- Next: a redirection to GitHub to authorize Travis CI as application.

Authorizing Travis as an application



Search GitHub Pull requests Issues Gist [redacted]

Authorize application

Travis CI by @travis-ci would like permission to access your account

Review permissions

Personal user data
Email addresses (read-only)

Repository webhooks and services
Read and write access

Commit statuses
Read and write access

Deployments
Manage deployments and deployment status

Organizations and teams
Read-only access

Organization access

Organizations determine whether the application can access their data.

✓

Authorize application

Travis CI

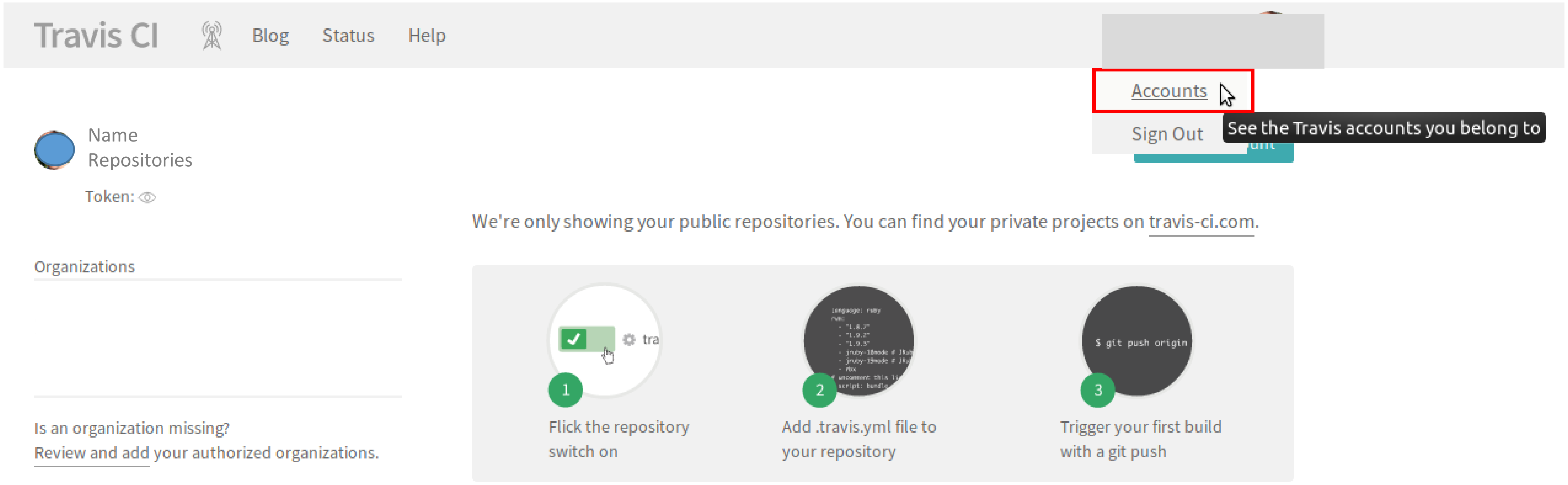
No description

Visit application's website

Learn more about OAuth

- After Travis CI is authorized it is listed as an application in Settings.
- Then Travic CI will check which repositories are available.

Adding a repository



Travis CI Blog Status Help

Accounts

Sign Out

See the Travis accounts you belong to

Name
Repositories

Token:

Organizations

Is an organization missing?
Review and add your authorized organizations.

We're only showing your public repositories. You can find your private projects on travis-ci.com.

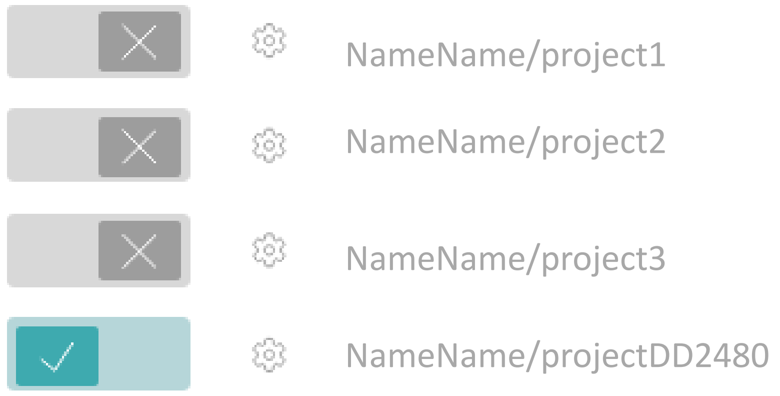
1 Flick the repository switch on

2 Add .travis.yml file to your repository

3 Trigger your first build with a git push

Activating a repository

- A repository can be activated by pressing the switch button.


NameName/project1
NameName/project2
NameName/project3
NameName/projectDD2480

Configuration

- Add *.travis.yml* file to your repository
- Minimum: specify the used programming language
- For Java, Travis supports Maven and Gradle as build system
- A minimal *.travis.yml* file for a Maven or Gradle project simply specifies the language:

```
language java
```

Configuration example

```
language: java
# use Java 8
jdk:
- oraclejdk8

# see https://blog.travis-ci.com/2014-12-17-faster-builds-
with-container-based-infrastructure
sudo: false

# cache the build tool's caches
cache:
  directories:
  - $HOME/.m2
  - $HOME/.gradle
```

Virtualisation evn. for build

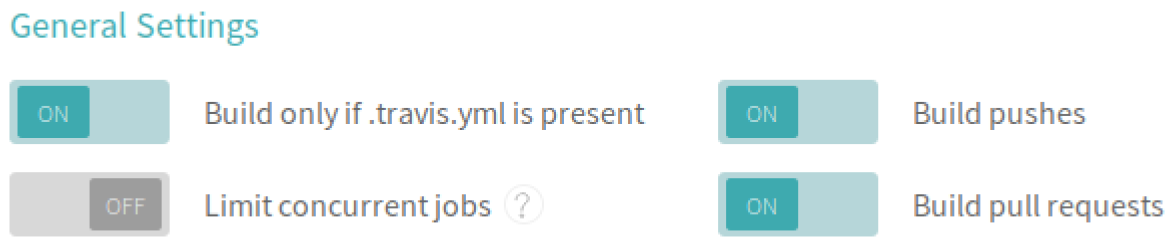
Caches lets Travis CI store directories between builds,
which is useful for storing dependencies that take longer to compile or download.

When a Gradle wrapper is available Travis CI will build your project by using `gradlew build` command

Upload all Gradle wrapper files, including the *gradle-wrapper.jar*

Triggering a build

- How a build is triggered can be configured in the *settings* section.
- You can enforce e.g., that
  - a *.travis.yml* is present to start a build
  - a build should be triggered at any time a push occurs or even if pull request comes



General Settings

ON	Build only if .travis.yml is present	ON	Build pushes
OFF	Limit concurrent jobs ?	ON	Build pull requests

- Here pushing a new commit will trigger a new build.

Other build configuration options

- In the build configuration you can also define other things, e.g.,
- Commands and scripts to be run before and after each build

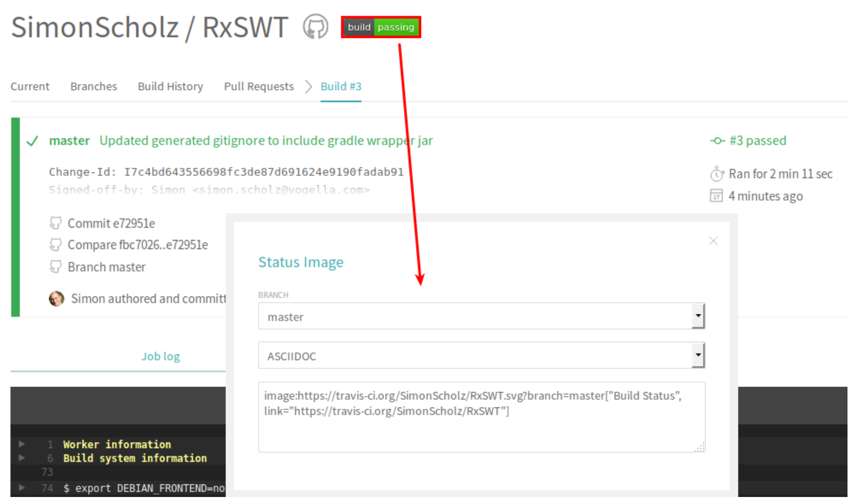
```
before_script:
   - git config --global user.name [myname]
```

- Notifications in terms of emails or chats alerts

```
notifications:
  email: false
  irc: "chat.freenode.net#travis"
```

Showing a status icon

- The build status determined by Travis CI is often shown in public.
- Click on the build status icon and choose from several options for embedding the status icon.



SimonScholz / RxSWT

✓ master Updated generated gitignore to include gradle wrapper jar

```
Change-Id: I7c4bd643556698fc3de87d691624e9190fadab91
Signed-off-by: Simon <simon.scholz@vogella.com>
```

Commit e72951e
Compare fbc7026..e72951e
Branch master
Simon authored and committ

#3 passed
Ran for 2 min 11 sec
4 minutes ago

Status Image

BRANCH: master

ASCIIDOC

```
image:https://travis-ci.org/SimonScholz/RxSWT.svg?branch=master["Build Status",
link="https://travis-ci.org/SimonScholz/RxSWT"]
```

```
Worker information
Build system information

$ export DEBIAN_FRONTEND=no
```

Webhooks: general idea

- Webhooks are automated messages sent from apps when something happens.
  - They have a message (*payload)*
  - are sent to a unique URL (app's address).
- When something happens, apps can push the data to each other and not waste their time checking and waiting.
- Jeff Lindsay: webhooks are "user-defined callbacks made with HTTP"
- Webhooks are data and executable commands sent from one app to another over HTTP

Webhooks in GitHub

- Webhooks allow you to build or set up GitHub Apps which subscribe to certain events on GitHub.com
- When one of those events is triggered, a HTTP POST payload sent to the webhook's configured URL.
- Can be used to update an external issue tracker, trigger CI builds, update a backup mirror etc.
- Webhooks can be installed on a specific repository.
- Webhook will be triggered each time one or more subscribed events occurs.

Webhooks: events

- You can configure a webhook, to choose which events you want to receive payloads for.
  - Subscribe to the useful for *you* specific events to limit the number of HTTP requests to your server.
- By default, webhooks are only subscribed to the *push* event
- You can change the list of subscribed events through the API or UI anytime
- E.g., if you subscribe to the `issue` event you receive *detailed payloads* every time an issue is opened, closed, labeled, etc.

Webhooks: payload format

- Each event type has a specific payload format with the relevant event information.
    - Push event has a more detailed webhook payload.
- Webhook payloads include the user who performed the event `sender`,`organisation` and `repository` which the event occurred on
- For a GitHub App's webhook may include the installation which an event relates to
- Idea of REST API