Continuous Integration mit TravisCI u.a.

Jan Steemann

Ziele für heute sind:

- eine kleine Applikation und Code-Änderungen daran automatisiert zu testen
- Build-Status und Coverage zu veröffentlichen
- die Code-Coverage der Tests zu ermitteln
- mehrere Test-Umgebungen abzudecken
- möglichst wenig Geld für Test-Infrastruktur auszugeben

Ausgangslage:

- eine (kleine) C++-Applikation zum Addieren von Zahlen
- Build-System ist CMake
- erste Version der Applikation kompiliert bereits unter Linux
- noch keine Tests vorhanden

Kompilieren:

- out of source-Build im Verzeichnis *build*:

```
mkdir -p build
(cd build && cmake .. && make)
```

Testen!

- zuerst brauchen wir ein Test-Framework!
- wir wählen Googletest (gtest)
- den Source-Code clonen wir direkt in unser Repository, Unterordner *tests*:

```
mkdir -p tests
cd tests
git clone \
https://github.com/google/googletest
```

Tests hinzufügen:

- nun kann der eigentliche Test-Code in *CalculatorTest.cpp* geschrieben werden
- anschließend kann der Test ins *CMakeFile* eingetragen werden und lokal getestet werden:

```
(cd build && cmake .. && make)
(cd build/tests && ctest -V)
```

Github service hooks

- wir wollen, dass alle Änderungen an der Applikation automatisch getestet werden
- Github bietet an, bei Code-Änderungen in einem Repository konfigurierbare externe Services aufzurufen
- diese können den aktuellen Code abrufen und z. B. Tests darauf ausführen

Integration mit TravisCI:

- als CI-"Dienstleister" wählen wir TravisCI (https://travis-ci.org/)
- zur Anmeldung bei TravisCI brauchen wir nur den Github-Account
- das Token unseres TravisCI-Accounts muss bei Github angegeben werden, damit Github an TravisCI "posten darf"
  (unter *Settings* > *Services*)

TravisCI-Build-Umgebung:

- Ubuntu 12.04 LTS
- mit vielen vorinstallierten Tools (z. B. automake, cmake, gcc/g++, clang), Skriptsprachen, Datenbanken
- die Build-Umgebung wird für jeden Build neu aufgebaut (clean state)

TravisCI-Build-Umgebung:

- die Default-Versionen in der Build-Umgebung passen oft nicht
- es können aber andere Versionen und auch zusätzliche Pakete installiert werden
- dadurch gibt es erst Unterstützung für C++11

.travis.yml:

- zur Konfiguration des Builds muss im Github-Repository unserer Applikation eine Datei *.travis.yml* hinterlegt werden
- diese Datei enthält die Build-Konfiguration und -Anweisungen für TravisCI
- darin müssen auch die eigentlichen *cmake*- und *ctest*-Anweisungen eingetragen werden

Wichtige Parameter in .travis.yml:

- *sudo*:
  true: legacy infrastructure mit Root-Rechten
  false: containerized infrastructure ohne Root
- *compiler*:
  z. B. g++ oder clang
- *addons*:
  zusätzliche zu installierende Pakete

Wichtige Parameter in .travis.yml:

- *install*:
  zur manuellen Installation zusätzlicher Pakete
- *script*:
  das eigentliche Build und Test-Kommando
- *after_failure*:
  Kommando nach fehlgeschlagenem Build
- *after_success*:
  Kommando nach erfolgreichem Build

TravisCI-Build-Status:

- bei jedem Push zu Github wird nun ein Build bei TravisCI angestoßen
- der Travis-CI-Build-Status ist entweder:

- das Build-Ergebnis ist öffentlich

TravisCI-Benachrichtigungen:

- nach jedem abgeschlossenen Build wird man bzgl. des Build-Status benachrichtigt, per Default-per E-Mail
- weitere Benachrichtigungsmöglichkeiten u.a.:
  - IRC
  - Slack
  - Gitter

Environment-Variablen:

- in einem Build-Skript können beliebige Environment-Variablen gesetzt werden
- diese können im Klartext oder verschlüsselt in die *.travis.yml* eingetragen werden, oder im Web interface von TravisCI hinterlegt werden
- mit Environment-Variablen und Konfiguration kann eine Build-Matrix erzeugt werden
- bestimmte Kombinationen können ausgeschlossen werden

Coverage:

- um Coverage hinzuzufügen, müssen die Anwendung und Tests mit Coverage-Support kompiliert und ausgeführt werden
- die Coverage-Ergebnisse können im Travis-Build mit */cov* prozessiert werden
- anschließend können sie automatisiert beim Dienst https://coveralls.io veröffentlicht werden

Windows-Tests:

- mittels Service-Integration von Appveyor kann die Applikation zusätzlich auf Windows gebaut und getestet werden
- zum Login bei appveyor.com wird ebenfalls der Github-Account benötigt
- nach dem Anmelden des Projekts wird eine Konfigurationsdatei *appveyor.yml* im Repository angelegt

Ergebnis:

- bei jedem Push in das Repository der Applikation werden die Tests auf Linux und Windows ausgeführt
- die Code-Coverage der Tests wird automatisiert berechnet
- es erfolgen automatisierte Benachrichtigungen über die Build-Status

Vor- und Nachteile:

- Abhängigkeit von externen Dienstleistern
- benutzte Dienste sind für Open source-Projekte kostenlos
- Build-Ergebnisse sind öffentlich
- Komponenten (TravisCI, Appveyor, Coveralls) sind nur lose verbunden
- Build-Konfiguration erfordert Skripting und Herumprobierei

Links:

- https://github.com/
- https://travis-ci.org/
- https://coveralls.io/
- https://www.appveyor.com/

TravisCI-Links:

- Travis-Lint (Validierung für .travis-yml-Dateien): https://docs.travis-ci.com/user/travis-lint
- Build-Status-Badges: https://docs.travis-ci.com/user/status-images/
- Build-Anpassung im Detail: https://docs.travis-ci.com/user/customizing-the-build/