

Improving Hosted Continuous Integration Services

Christopher Weyand, Jonas Chromik, Lennard Wolf,
Steffen Kötte, Konstantin Haase, Tim Felgentreff,
Jens Lincke, Robert Hirschfeld

Technische Berichte Nr. 108

des Hasso-Plattner-Instituts für
Softwaresystemtechnik
an der Universität Potsdam



Technische Berichte des Hasso-Plattner-Instituts für
Softwaresystemtechnik an der Universität Potsdam

Christopher Weyand | Jonas Chromik | Lennard Wolf | Steffen Kötte |
Konstantin Haase | Tim Felgentreff | Jens Lincke | Robert Hirschfeld

Improving Hosted Continuous Integration Services

Bibliografische Information der Deutschen Nationalbibliothek

Die Deutsche Nationalbibliothek verzeichnet diese Publikation in der Deutschen Nationalbibliografie; detaillierte bibliografische Daten sind im Internet über <http://dnb.dnb.de/> abrufbar.

Universitätsverlag Potsdam 2017

<http://verlag.ub.uni-potsdam.de/>

Am Neuen Palais 10, 14469 Potsdam

Tel.: +49 (0)331 977 2533 / Fax: 2292

E-Mail: verlag@uni-potsdam.de

Die Schriftenreihe **Technische Berichte des Hasso-Plattner-Instituts für Softwaresystemtechnik an der Universität Potsdam** wird herausgegeben von den Professoren des Hasso-Plattner-Instituts für Softwaresystemtechnik an der Universität Potsdam.

ISSN (print) 1613-5652

ISSN (online) 2191-1665

Das Manuskript ist urheberrechtlich geschützt.

Druck: docupoint GmbH Magdeburg

ISBN 978-3-86956-377-0

Zugleich online veröffentlicht auf dem Publikationsserver der Universität Potsdam:

URN <urn:nbn:de:kobv:517-opus4-94251>

<http://nbn-resolving.de/urn:nbn:de:kobv:517-opus4-94251>

Preface

Developing large software projects is a complicated task which can be demanding for developers. Continuous integration (CI) is a common practice involving frequent testing and integration of changesets to keep them small and hence easily comprehensible. With CI, programmers can reduce complexity and effort. Travis CI is a service that offers continuous integration and continuous deployment. Software projects are built, tested, and deployed using the Travis CI infrastructure without interrupting the development process. This report describes the architecture of Travis CI, presents how periodic building is implemented, proposes a feature addition for dependent builds, and discusses ways of visualizing data generated by the CI process.

July 2016

The Authors

Große Softwareprojekte zu entwickeln, ist eine komplizierte Aufgabe und fordernd für Entwickler. Kontinuierliche Integration ist eine verbreitete Praxis zur Komplexitätsreduktion. Durch häufiges Integrieren und Testen werden Änderungen klein gehalten und sind daher übersichtlich. Travis CI ist ein Dienst, der kontinuierliche Integration und kontinuierliche Bereitstellung anbietet. Softwareprojekte werden durch die Travis CI Infrastruktur gebaut, getestet und bereitgestellt, ohne dass der Entwicklungsprozess unterbrochen wird. Dieser Bericht beschreibt Architektur und Funktionsweise von Travis CI, zeigt wie periodisches Bauen implementiert wurde, schlägt eine Funktionserweiterung für Buildsequenzen entlang von Abhängigkeitsbäumen vor und analysiert, wie CI-Daten visualisiert werden können.

Juli 2016

Die Autoren

Contents

1	Introduction	1
2	The Architecture of Travis CI	3
2.1	Introduction	3
2.2	Background	3
2.3	Approach	5
2.4	Travis Implementation	9
2.5	Evaluation	25
2.6	Future Work and Conclusion	28
3	Interfaces for Expressing Periodic Task Schedules	29
3.1	Introduction	29
3.2	Context	29
3.3	Problem	31
3.4	Approach	33
3.5	Implementation	44
3.6	Evaluation	49
3.7	Related Work	52
3.8	Conclusion	56
4	Dependency Management for Hosted Continuous Integration Services	59
4.1	Introduction	59
4.2	Background	60
4.3	Motivation	63
4.4	Approach	65
4.5	Implementation	74
4.6	Evaluation	81
4.7	Related Work	82
4.8	Future Work	83
4.9	Conclusion	84

5	Visualizing Build Data in Continuous Integration Services	85
5.1	Introduction	85
5.2	State of the Art	85
5.3	Problem	91
5.4	Approach	92
5.5	Implementation	96
5.6	Evaluation	103
5.7	Related Work	104
5.8	Future Work	106
5.9	Conclusion	107
6	Conclusion	109

1 Introduction

Continuous integration (CI) is an important practice in agile software development. Since CI infrastructures are usually very sophisticated and complex to set up, hosted continuous integration services have become popular in the past years. Travis CI is one such service for projects hosted on GitHub.

Although Travis CI offers fully functional continuous testing and deployment for many programming languages and platforms, it still lacks some features common to other CI tools. In particular, a way of coping with dependencies, as well as a web page presenting useful information about the software project are missing. These issues are addressed in this report.

We present and analyze the structure of the micro-service architecture of Travis CI, as well as the collaboration of the individual components. Using this knowledge we propose two different approaches to dealing with software dependencies. The first approach is building and testing projects periodically. Problematic changes in dependencies will lead to failing tests, causing Travis CI to inform the developers about the failure. The second approach introduces an interface for modeling dependency relationships in graphs, along which build sequences can run. With these, projects can be built whenever its dependencies were modified. For creating a web page presenting information about the software project, we analyzed which information is useful for software developers and project managers. We propose charts and data to be displayed on such a web page.

This report is structured as follows: Chapter 2 describes the micro-service architecture of Travis CI. Chapter 3 proposes different means of expressing schedules for periodic building. Chapter 4 presents the interface for dependency graphs and the build sequences resulting from them. Chapter 5 analyzes which charts and data are useful for developers and introduces an overview page that combines them.

2 The Architecture of Travis CI

Travis CI is a hosted continuous integration service structured as a microservice architecture. The modular components, although minimally coupled, are complex subsystems running in the cloud. Due to Travis' utilization of intertwined software libraries and sophisticated programming mechanics, an exhaustive understanding of the system's construction is nontrivial. Providing the basis for an evaluation of the architecture's modularity, extensibility, and scalability, this chapter describes the overall architecture as well as the most crucial subcomponents.

2.1 Introduction

The hosted continuous integration service Travis CI helps software developers verify that new changes are correctly integrated into the main source of files under version control. It therefore executes automated tests on virtual machines running in the cloud and informs the developer about the results. Built as a microservice architecture, Travis is limited to the repository hosting service GitHub, where all of Travis' modules are also developed.

For Travis users concerned with security issues or open source developers trying to improve Travis, it is crucial to understand Travis' system as a whole as well as the most important subcomponents. However, achieving an exhaustive understanding of the system only by reading the documentation and the source code requires disproportional effort. This is due to the large number of external dependencies and libraries, the years of experience the developers put into Travis, and the intrinsic complexity of a microservice architecture [4, p. 49]. A comprehensive description of the system would ease improvements and other work related to the Travis system. Therefore this chapter describes the architectural approach in general and the implementation of each submodule in detail.

2.2 Background

Continuous integration (CI) is a common practice of software engineering [5, 7]. Applying continuous integration presupposes that the developer team maintains the source files under version control. Every developer creates a working copy of the source files, changes them, and finally integrates his work back into the mainline of the shared repository. The longer a working copy is separated, the more it diverges from the mainline. To prevent merge conflicts the integration is done at least daily [19]. Automated tests verify the success of the integration process. Therefore continuous

integration encourages exhaustive testing or even test driven development. The tools and services to perform these tests run asynchronously to reduce time and effort for the developer. Additionally they set up a clean test environment, because every previous change that could affect the test execution makes it more difficult to identify errors.

Historically CI services ran on a local setup. Direct access to the machines running the tests is secure and the process is easily monitorable. Most local services offer extensive support for plugins, scripts, and all means of customization. While agile software development became more common, the cloud based services provided a simpler interface and did not need in-house computing power. They suffice for the average customer's needs, in trade for complex configuration settings and access to running hardware. Nevertheless, large companies still prefer local testing servers because of security concerns.

A popular open source tool for continuous integration on a local setup is Jenkins.¹ There are many similar tools, such as Strider CD² or GoCD³ (developed by ThoughtWorks, where Martin Fowler did research on continuous integration).

A cloud based approach can provide more comfort and simplicity for the user. Most cloud based CI services just require access to the repository hosting service and a simple configuration file within the source repository. Since the tests either run on servers hosted by the CI service or on virtual machines in the cloud, neither own hardware nor maintenance are required. The hosted CI service can manage builds on different operating systems as well as concurrent builds. Every service comes with its own advantages and drawbacks. AppVeyor⁴ unlike most other services, features Windows, MagnumCI⁵ supports GitHub, GitLab, and Bitbucket as repository hosting services, Codeship⁶ uses Docker, SnapCI⁷ represents ThoughtWorks cloud complement to GoCD, and CloudBees⁸ resembles a cloud based Jenkins. However, the most popular⁹ hosted continuous integration service is TravisCI.¹⁰ Inspired by a hosted CI service for Ruby and Rails projects called RunCodeRun,¹¹ Travis developed from a small idea and the work of a few developers [21] into a global, successful company.

¹<https://jenkins.io/> (visited on 2016-06-13).

²<https://github.com/Strider-CD/strider> (visited on 2016-06-13).

³<https://www.thoughtworks.com/go/> (visited on 2016-06-13).

⁴<http://www.appveyor.com/> (visited on 2016-06-13).

⁵<https://magnum-ci.com/> (visited on 2016-06-13).

⁶<https://codeship.com/> (visited on 2016-06-13).

⁷<https://snap-ci.com/> (visited on 2016-06-13).

⁸<https://www.cloudbees.com/> (visited on 2016-06-13).

⁹In fact the GitHub developer documentation on building a CI service mentions Travis and Jenkins as the alternatives to building it yourself.

¹⁰<https://travis-ci.org/> (visited on 2016-06-13).

¹¹<http://thinkrelevance.com/blog/2008/10/14/runcoderun-now-open-to-the-anonymous-public-2> (visited on 2016-06-13).

2.3 Approach

In this Section, we will describe Travis' service and its structure in general. We introduce the central modules of the microservice architecture and divide them between the primary build sequence (life cycle of accepting, evaluating, and executing) and the secondary post execution steps. The former includes the necessary modules to run a test while the latter represents all additional steps during and after the execution.

Travis is a hosted continuous integration and continuous deployment service for the repository hosting service GitHub. Its main purpose is the execution of test in so called builds and the optional deployment of the tested software. A build is created each time a developer pushes a change to the shared repository. A build includes one or more jobs, which set up the target project and execute the tests. Different programming languages are supported for the target project.

Their build flow is shown in Figure 2.1.

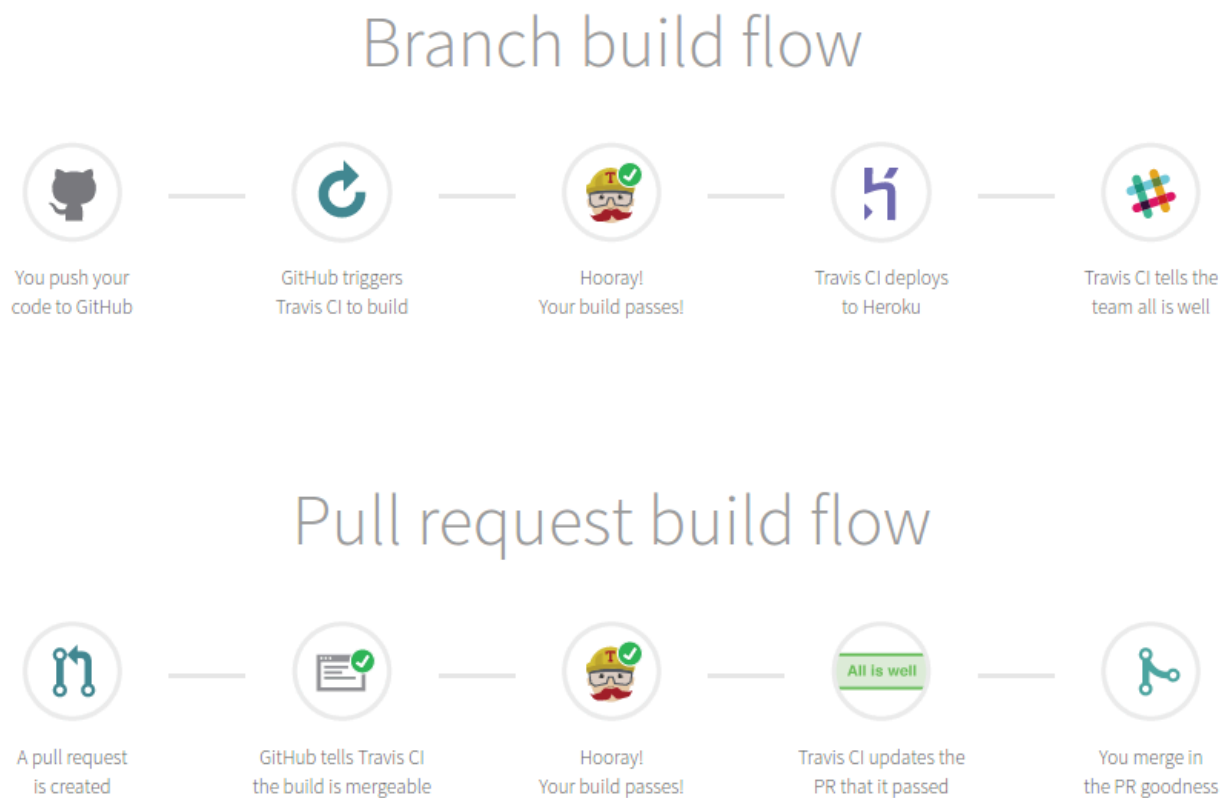


Figure 2.1: Travis Build Flow taken from travis-ci.org

Listing 2.1: A simple configuration file for a Smalltalk repository

```
language: smalltalk
smalltalk:
  - Squeak5.0
  - Squeak4.6
os:
  - linux
  - osx
```

There is a website to show the build results to the user.¹² The Travis website provides information about all previous builds and jobs as well as the settings for each repository. The logs of a running job can even be viewed as they get generated. Additionally there exists a command line client as a ruby gem. Both, Travis Web and Travis CLI access their data through the public Travis API.¹³

A configuration file¹⁴ specifies the programming language, the operating system to use during testing, and the deployment options. Furthermore the user can add custom scripts and configure most execution steps [46]. Eventually each build leads to a matrix of jobs including all combinations of different expandable parameters.

As an example the file in Listing 2.1 would result in a build with four jobs: Squeak5.0 and Squeak4.6 each on Linux and on OS X.

2.3.1 Life Cycle of Accepting, Evaluating, and Executing

The life cycle of accepting, evaluating, and executing a build request is the core of Travis' functionality from accepting the GitHub event to the execution of software tests. It starts, when a user pushes a source code change to GitHub or a pull request is opened.

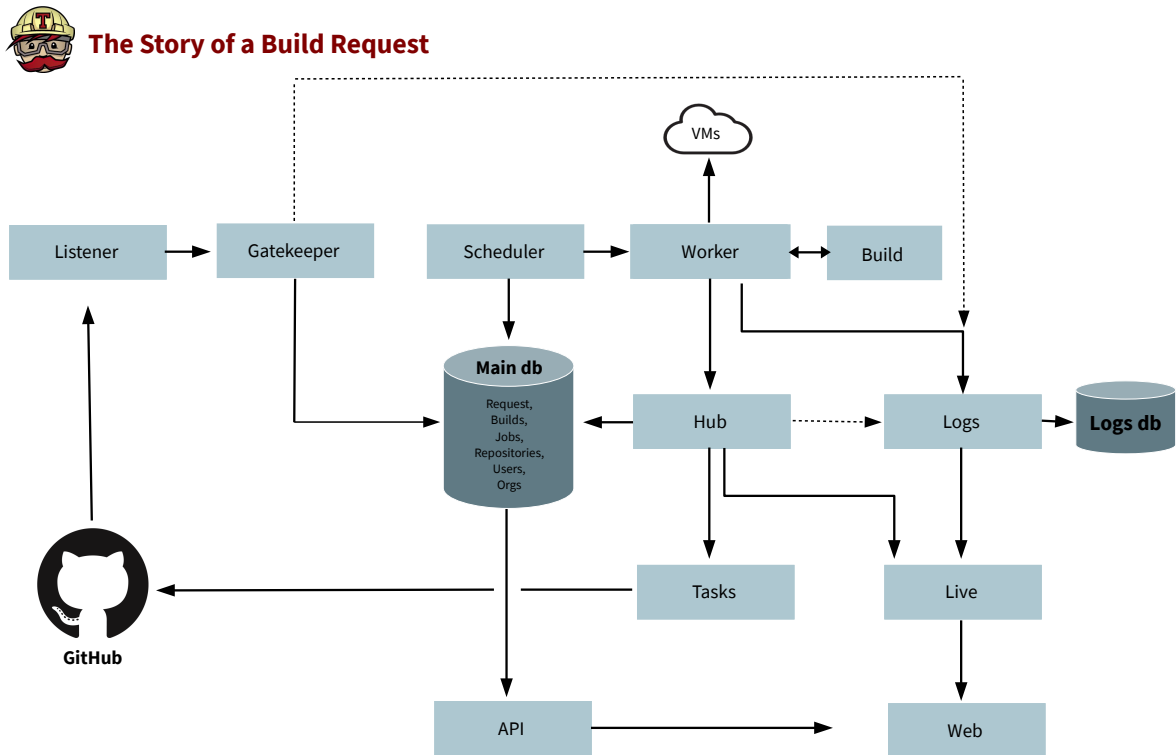
Listener Responsible for listening to these changes is Travis Listener. GitHub offers a hook to inform the Listener or other services about events. The Listener creates a build request and starts the build pipeline to process the recognized action.

Gatekeeper After the Listener created the build request the Gatekeeper picks it up. Gatekeeper approves and configures the build request. By fetching the configuration file (`.travis.yml`) from the GitHub repository, it can create a build entry in the main database. The matrix yields one or more jobs, which are associated with the build record. All those jobs are initially in a "created" state, awaiting execution.

¹²<https://travis-ci.org> (visited on 2016-06-16). The page for Travis CI Enterprise is available under <https://travis-ci.com> (visited on 2016-06-15).

¹³<https://api.travis-ci.org> (visited on 2016-06-03).

¹⁴Travis' configuration files must be called `.travis.yml` and placed into the repository root.



v 1.1

Figure 2.2: The Story of a Build Request made by Travis

Scheduler The third element in the life cycle of accepting, evaluating, and executing a build request is Travis Scheduler. The Scheduler continuously fetches all jobs in the “created” state from the main database. Then it groups them by their owner, which is a GitHub organization or a single user. The Scheduler evaluates the amount of jobs to be queued for that owner. Every user and organization has a limit of jobs that are allowed to run concurrently. For example if there are three running jobs and the developer’s concurrency limit is five, it queues two more jobs. To queue a job means to signal that it is ready to be executed by a Worker.

Worker Travis Worker is the final element of the primary life cycle. Travis runs multiple Worker instances simultaneously to handle the high load on the system. They start virtual machines (VM’s) on different cloud providers like Amazon EC2 and MacStadium and run a shell script on the VM. The script contains the setup of the target project and the tests for a job. Once the script is finished the Worker is ready to execute the next job.

2.3.2 Post Execution Steps

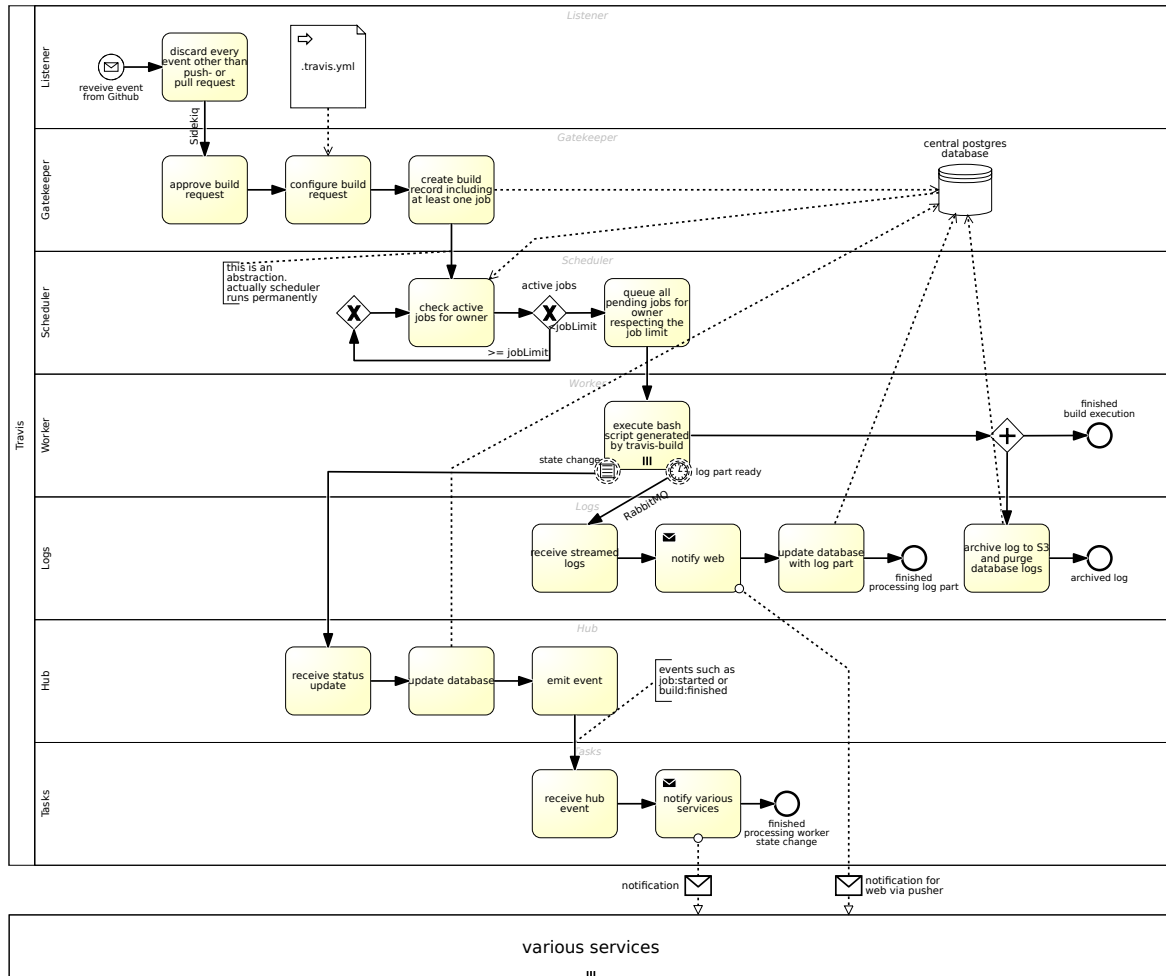


Figure 2.3: Travis' central Business Process

Travis' work is not done with the execution of the tests by Worker. For users to be able to access relevant data about their tests, Travis needs to save logs, maintain a database, send out notifications via various services, provide a way to access this data, and possibly execute a deployment step if the build was successful. There are various deployment options¹⁵ like Heroku, Launchpad, or NPM.

To realize these functions Travis uses the *Hub*, *Logs* and *Tasks* modules. *Travis Web* and *Travis CLI* furthermore provide access to the required data, using *Travis API*. The deployment is done by the Worker after it finishes a successful build. Travis maintains a Ruby Gem called *Dpl* to support the deployment to other providers.

¹⁵<https://docs.travis-ci.com/user/deployment/> (visited on 2016-06-15).

Hub Each time the state of the job changes Worker sends a message that will be processed by Travis Hub. Hub then changes the job's state in the database and sets the attributes concerning that state change. If all jobs of a build are finished, it sets state of the corresponding build as well. Also Hub emits events like `build:finished` for the Tasks module.

Logs During execution Worker streams the log to Travis Logs. The log is streamed in parts, which are temporarily stored in the database and can be assembled to the complete log on demand. When the job is finished, Logs archives the complete file to Amazon S3 and the database records are purged.

Tasks Tasks receives events from Hub and sends notifications to the user via E-Mail, Slack etc. It also updates the Web UI of Travis and GitHub.

2.4 Travis Implementation

This Section separately describes each major component of Travis architecture introduced in Section 2.3.1 or 2.3.2. In order of participation in the build processing, these components are Listener, Gatekeeper, Scheduler, Worker, Hub, Logs, and Tasks. Unlike the others, Gatekeeper is not open source and thus omitted in this section. All information presented in this section is based on Travis' source code and readme files, which are accessible via GitHub.

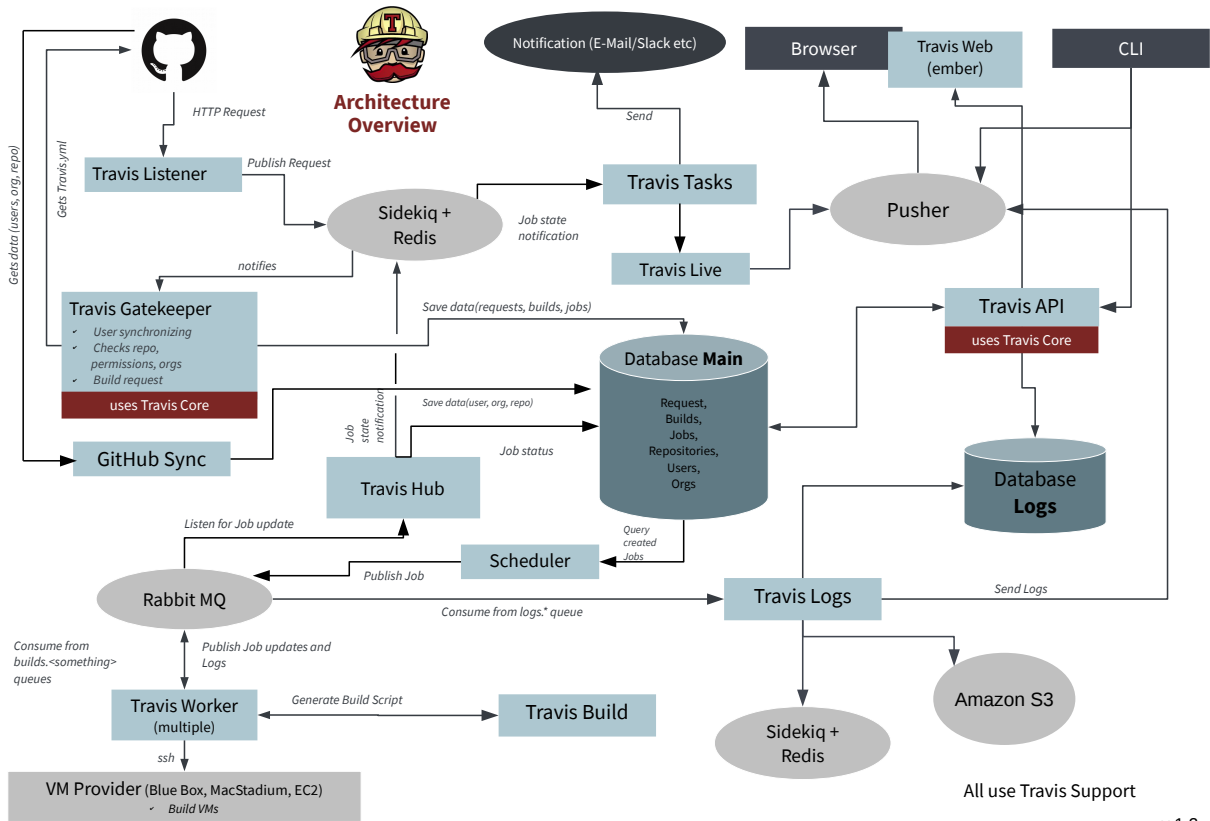
Travis is implemented as a microservice architecture, which parts are deployed to Heroku. In addition to a GitHub repository for each service, Travis has multiple shared helper repositories. For example *travis-migrations* holds all database migrations, *travis-core* contains code for Gatekeeper and API, and *travis-ci* acts as a central issue tracker. Figure 2.4 shows a detailed overview, which is relevant for all Subsections.

2.4.1 Travis Listener

Travis Listener comes first in the build pipeline. Triggered by a user action, GitHub informs the Listener about the event. If the action should start the execution of a build, i.e. is a push or pull event on GitHub, Listener notifies Gatekeeper. Written in Ruby (as most Travis modules are), this Sinatra application discards all other event types. The notification send to Gatekeeper is called a build request and is transferred using Redis and Sidekiq (see Figure 2.5).

Listener utilizes a GitHub Webhook, Raven, Metriks and Rack, which Sidekiq depends on. The Listener is deployed on Heroku with Unicorn.

The predefined GitHub Webhook listens for GitHub events and sends a POST request with a payload to the Listeners URL as configured. The URL is `http://notify.travis-ci.org`. A default GitHub Webhook processes only push events. Although the listener discards most of it, the GitHub service sends push, pull request, issue comment, public, and member events as seen in Listing 2.2. A public event is



v 1.2

Figure 2.4: Architecture Overview made by Travis

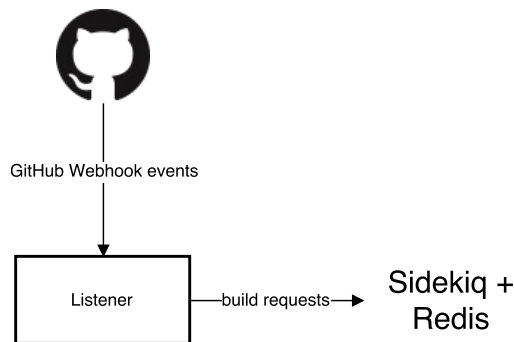


Figure 2.5: Travis Listener Overview

Listing 2.2: github/github-services/lib/services/travis.rb

```

default_events :push, :pull_request, :issue_comment, :public, :member

[...]

def receive_event
  http.ssl[:verify] = false
  http.basic_auth user, token
  http.headers['X-GitHub-Event'] = event.to_s
  http.headers['X-GitHub-GUID'] = delivery_guid.to_s
  http_post travis_url, :payload => generate_json(payload)
end

```

triggered by a private repository becoming open source and a member event signals that a new user was added as a collaborator.¹⁶

As the Listener's core functionality is to listen on POST requests, the application is a subclass of Sinatra::Base. On a POST request, if the payload exists and the IP is valid, it calls the `handle_event` method. Finally, if the event type is push or pull_request a build request is published using the Sidekiq queue "build_requests" to which the Gatekeeper Service subscribes. The `handle_event` method is shown in Listing 2.3. Discarding all events except for push events or pull request is done at line 80 with the `handle_event?` check.

Listing 2.3: travis-listener/lib/travis/listener/app.rb

```

79 def handle_event
80   return unless handle_event?
81   debug "Event payload for #{uuid}: #{payload.inspect}"
82   log_event(event_details, uuid: uuid, [...] , repository: slug)
83   Travis::Sidekiq::BuildRequest.perform_async(data)
84 end
85
86 def handle_event?
87   settings.events.include?(event_type)
88 end

```

2.4.2 Travis Scheduler

After Gatekeeper created the database records for a build and its jobs, Travis Scheduler identifies pending jobs in the database and queues them. To queue a job means to assign it to a waiting Worker. All pending jobs, meaning that they are in *created* state, are queued with respect to their owner's concurrency limit. For open source projects the default limit is five. An overview of the Scheduler is shown in Figure 2.6.

¹⁶<https://developer.github.com/v3/activity/events/types/>
(visited on 2016-06-10).

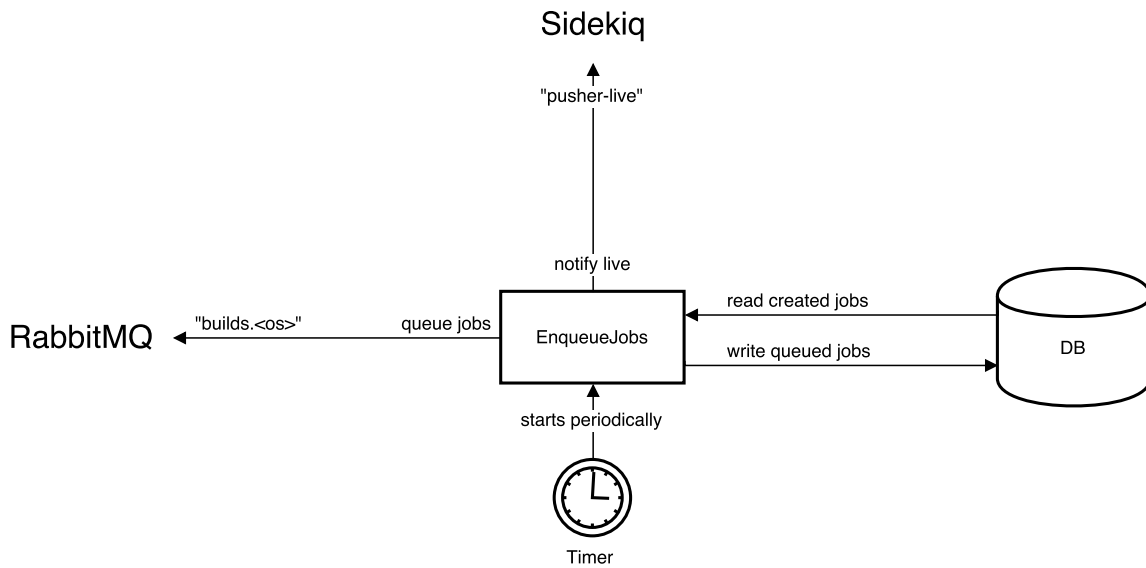


Figure 2.6: Travis Scheduler Overview

Initialization

When Scheduler is deployed, it executes the methods `setup` and `run`. Listing 2.4 shows the implementation of these methods. Setup calls multiple other setup methods to configure RabbitMQ, the database, exception handling with Raven, Metriks, Sidekiq and feature flags for the application. There is a separate Travis repository for AMQP¹⁷ which is included in the gemspec. The Schedule class therefore requires `travis/amqp` to call `Travis::Amqp.setup(config.amqp.to_h)`. After setting up the client for RabbitMQ, the scheduler connects to the database. The configuration either refers to `travis_development` or `travis_production` as database depending on the environment. Metriks is used throughout the scheduler to measure execution times. Finally the scheduler configures Sidekiq to communicate with Redis and the last setup method prepares to handle feature flags using Rollout.

Subsequently the Scheduler starts its work by entering an endless main loop that is executed periodically in intervals specified in Scheduler's configuration file. The method `enqueue_jobs_periodically` starts the enqueueing after acquiring a lock on the database. Like most steps performed by the Scheduler, this is measured by Metriks.

Listing 2.4: `travis-scheduler/lib/travis/scheduler/schedule.rb`

```
def setup
  Travis::Amqp.setup(config.amqp.to_h)
  Travis::Database.connect(config.database.to_h)
  Travis::Exceptions::Reporter.start
  Travis::Metrics.setup
end
```

¹⁷<https://github.com/travis-ci/travis-amqp> (visited on 2016-06-10).

```

Support::Sidekiq.setup(config)
Support::Features.setup(config)

declare_exchanges_and_queues
end

def run
  enqueue_jobs_periodically
end

```

Enqueue Jobs

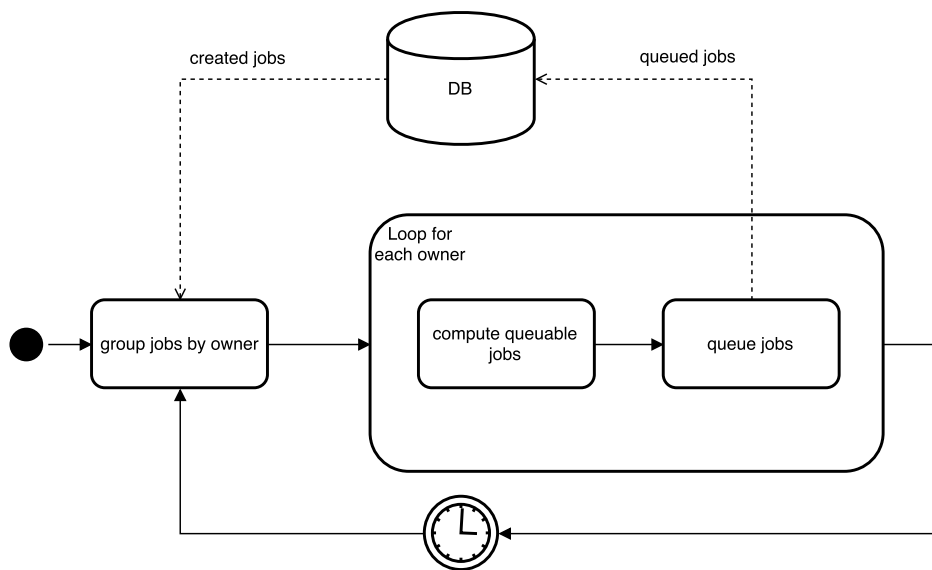


Figure 2.7: Activity Diagram for the EnqueueJobs Class

The periodic execution begins with the instantiation of the EnqueueJobs class, which contains the actual queuing code (see in Figure 2.7). Initially that class groups all jobs by owner and enters a loop over all owners and their jobs. After computing how many jobs are queueable for each owner up to their specific concurrency limit, the queueable jobs are enqueued as seen in Listing 2.5.

To queue a job, three things have to be done:

1. publish the event to RabbitMQ for the workers to pick up,
2. update the database,
3. notify Travis Live about the change so the Web UI can be updated.

To achieve that, the EnqueueJobs service utilizes AMQP to publish to RabbitMQ, updates the job attributes directly in the database via the Active Record model (which includes setting `queued_at` to the current time), and calls a helper method to make

Listing 2.5: travis-scheduler/lib/travis/scheduler/services/enqueue_jobs.rb

```
def enqueue_all
  grouped_jobs = jobs.group_by(&:owner)
  [...]
  grouped_jobs.each do |owner, jobs|
    next unless owner
    [...]
    limit = strategy.new(owner, jobs)
    Travis.logger.info "About to evaluate jobs for: #{owner.login}."
    queueable = limit.queueable
    [...]
    enqueue(queueable)
    [...]
  end
end
```

Sidekiq push the payload to the “pusher-live” queue. Actually the architecture visualization (Figure 2.4 in Section 2.4) does not show the connections for item 2 and 3 as it is not as detailed as the individual section for each repository.

2.4.3 Travis Worker

Worker¹⁸ is a command line client that runs the Travis CI jobs. It uses VM providers to execute the script, which the *Build* module generates on demand (see Figure 2.2). While the Worker goes through the different stages of execution, he publishes state and log updates to RabbitMQ. The state updates are handled by Travis Hub and the streamed logs by Travis Logs.

Internal Structure

Figure 2.8 shows the architecture of Worker. The application maintains a processor pool. Processors can be added or removed from the pool. On startup a new CLI object is created by the main function, which then calls Setup and Run. Figure 2.9 illustrates the order in which messages are send. It also shows, where communication is done synchronous or asynchronous. This distinction is a crucial aspect of the Worker.

Setup During setup the CLI object creates a configuration from the CLI context, initializes Sentry and Metrics (Go’s equivalent for Metriks), and creates the job queue, canceller, logger and build script generator. Additionally it determines the backend provider according to the configuration, which it sets up immediately. In the end, the processor pool is created. The job queue is either *amqp* or *file*. The former connects to RabbitMQ as shown in Figure 2.4, whereas the latter is meant for local execution

¹⁸Unlike other Travis repositories it is not following the naming convention “travis-ci/travis-*”. Since the name “travis-worker” was taken by the old deprecated repository, the current one is just called “worker”.

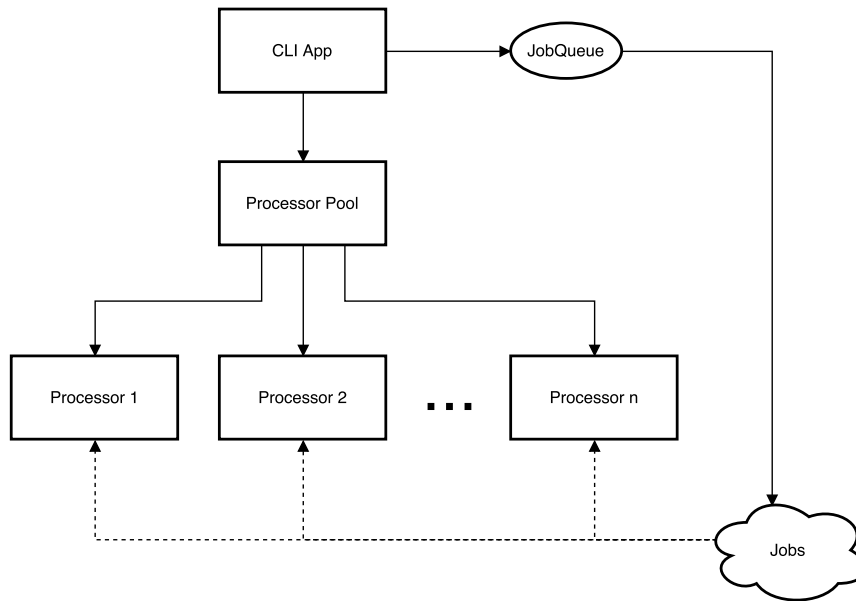


Figure 2.8: Worker Overview

of the Worker. There are separate implementations of the Job, JobQueue, LogWriter, and Cancellor interface prefixed with *amqp* or *file*.

Run Go features a keyword to fork a function from the running process. Asynchronous processes communicate through channels. The CLI uses this behavior to establish a signal handler. The handler listens for commands to shut the worker down, print information about all processors and increase or decrease the size of the processor pool.

The loop in Figure 2.9 shows how the processors are created. A wait group is used to synchronize all created processors before returning from Run. `Incr` creates a new processor and increments the wait group. The processors are run asynchronously as seen in Figure 2.9 indicated by the pointed arrowhead. Also the keyword `defer` is used to decrease the wait group after the processor terminates. This Go feature allows to delay the execution of a command to after the current method is finished. `Decr`, as the opposite to `Incr`, shuts one processor down.

Job Execution

In idle state each process waits for commands on different channels. There is a job channel providing the process with incoming jobs. As soon as the process receives a job, it executes the steps shown in Figure 2.10. A step is implemented as a class responding to the `Run` and `Cleanup` methods, which receive a `StateBag` as input to carry information over to the next step.

Subscribe Cancellation The canceller is basically a map from job IDs to channels. To subscribe to it means to receive a notification through that channel when the can-

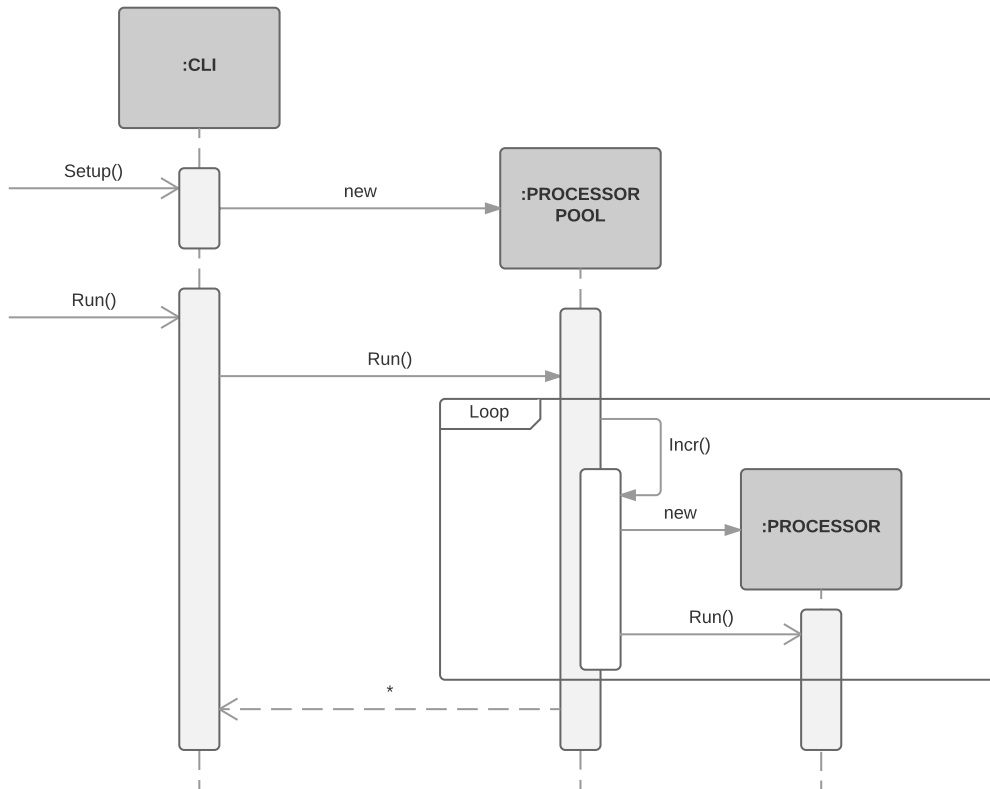


Figure 2.9: UML Sequence Diagram



Figure 2.10: Steps performed by Worker during Job Execution

celer gets the cancel command from RabbitMQ. The queue is named “worker.commands”. This step creates a channel and subscribes to the canceler with the ID of the currently processed job. The channel is saved in the StateBag.

Generate Script The second step represents the connection between Worker and Build as shown in Figure 2.4. It uses the build script generator to generate the script. Since the generator was configured earlier during the CLI setup, this step just calls the Generate method, which sends a HTTP request to the generator URL. Usually Travis Build then answers the requested script. Finally the script is saved in the StateBag.

Send Received Send Received is a very simple step. It updates the job state to *received* and sets the *received_at* attribute to the current time. The state update

is published to RabbitMQ in case of the *amqp* job. Listening to the “reporting.jobs.builds” queue, Hub picks up this event and changes the state in the database.

Start Instance At this point during job execution a virtual machine needs to be started. The previously configured backend provider receives the signal to start an instance according to the job’s attributes. In case of an error while starting the VM the job is requeued. This means that the Worker stops execution and the job can now be picked up by a new Worker.

Upload Script The build script generated by Travis Build in step 2 is now uploaded to the VM instance. The StateBag allows access to the script and the instance, so errors could only occur during the upload and would lead to a requeue of the job, like in the last step.

Update State Although this step is the shortest one, its cleanup involves setting the job’s state according to the result of the build script including the *finished_at* attribute. On initial execution, this step updates *started_at* and publishes the *started* state change to RabbitMQ, analogous to Send Received.

Open Log Writer Even though it may sound trivial to create a log writer, especially because it only executes an existing helper function, it is done in a separate step. The individuality of this stage depends on the fact that opening a log writer could produce an error. If so, the job needs to be requeued.

Run Script Finally the run script step uses all prepared resources of previous steps to execute the build script on the VM instance. It asynchronously starts a function to run the script. All output the script produces is redirected to the log writer. The log writer periodically flushes its content. This means that it publishes log parts to the RabbitMQ queue “reporting.jobs.logs”. After running the script the final result gets directed into the result channel. During execution Go’s *select* statement is used to wait on multiple channels (see Listing 2.6). There are four possible events that can occur while waiting for the build result: it finishes execution indicated by the result channel, the context cancels the job, the job was canceled by the user or the log writer timed out. A signal from the result channel could also be an error during execution. The context cancels, if the maximum time for a job is exceeded. If a user cancels a job, this message is send to the worker using RabbitMQ. This is recognized by the canceler mentioned at step one. The log writer times out, if no output is produced in a large amount of time, hence the corresponding log message states that ‘this potentially indicates a stalled build or something wrong with the build itself’¹⁹

¹⁹https://github.com/travis-ci/worker/blob/master/step_run_script.go#L92 (visited on 2016-06-14).

Listing 2.6: Relevant Channels waited on during Job Execution

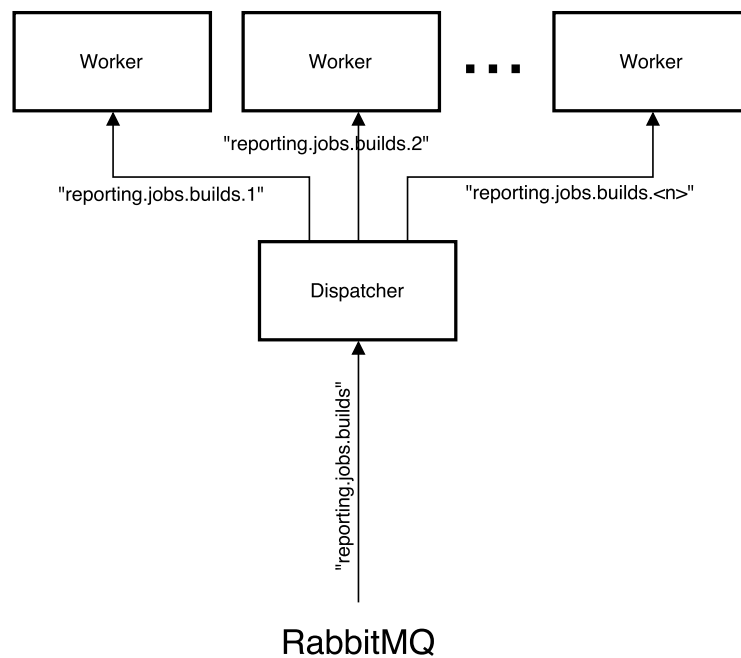
```

select {
case r := <-resultChan:
  [...]
case <-ctx.Done():
  [...]
case <-cancelChan:
  [...]
case <-logWriter.Timeout():
  [...]
}

```

2.4.4 Travis Hub

Travis Hub is a relatively small service written in Ruby. Its main purpose is to listen to state updates from multiple Worker instances, which were discussed in Section 2.4.3. Receive, start, finish, cancel, and restart are the possible events in order of occurrence. Eventually each of these events leads to a database update of the job or even the corresponding build. Additionally Hub handles featured addon services and notifies Travis Tasks to run the corresponding task.

**Figure 2.11:** Dispatcher

Dispatcher

As shown in Figure 2.11, the Hub consists of a dispatcher and multiple workers, not to be confused with the Worker in Section 2.4.3. Therefore the Hub's components will not be capitalized. Also there is a drain component, that just passes incoming events to a Sidekiq queue called "hu". Due to its trivial nature Figure 2.11 does not include the drain.

The default RabbitMQ queue for job updates is "reporting.jobs.builds". Ideally the dispatcher assigns incoming events to the workers using a *Round Robin* scheduling algorithm. This implies that events are dispatched in equal portions and in circular order. As it circles through the workers using the job's `source_id` modulo the amount of workers, the order can get mixed up, though. The code for this looks as follows: `key = job.source_id % count + 1`. Computing this number results in an individual queue for each responsible worker, which is `reporting.jobs.builds.<key>`. Thus the first worker subscribes to `reporting.jobs.builds.1`, the second to `reporting.jobs.builds.2`, and so on. The dispatcher publishes the event to this queue, meaning that Hub uses RabbitMQ for internal communication.

This concept allows the Hub to run in a *solo* mode. All dispatcher and workers are replaced by a single solo-worker instance, which listens directly to events on the main queue, rather than the queues appended with the worker's number. In fact all workers used with a dispatcher inherit from *solo*.

Worker

A worker handles events by creating a handler (see Figure 2.12). This handler either reroutes the event or handles it. To handle an event means to update the database with the new state. Also updating a job or a build notifies handler for featured addon services as shown in Figure 2.12. All *cancel* events are published to the "worker.commands" queue, causing the canceler mentioned in Section 2.4.3 to cancel a running job.

Internally the services `updateJob` and `updateBuild` are used, which perform three steps during their runtime. As seen in Listing 2.7 and 2.8, the steps are *validate*, *notify* and *update_job* or *update_jobs* respectively. Furthermore the services use a distributed locking mechanism based on Redis that acquires an exclusive access to the job or build being processed. *Validate* checks if the current event is included in the list of valid events²⁰ *Notify* publishes a cancel command to the "worker.commands" queue if the current event is *cancel*. This is symbolized by the bottom right arrow to RabbitMQ and is not to be confused with the "notify" arrow which leads from the update service to each handler (see Figure 2.12). In fact, the `update_job/update_jobs` method notifies the handler. Nevertheless this method primarily updates the database records and possibly adds attributes like `started_at` or `finished_at` associated with the time of the change. The Active Record model is responsible for managing the states of a job or build. If the last job of a build finishes, the corresponding build needs to be updated as well, so jobs propagate events to their build.

²⁰receive, start, finish, cancel, and restart.

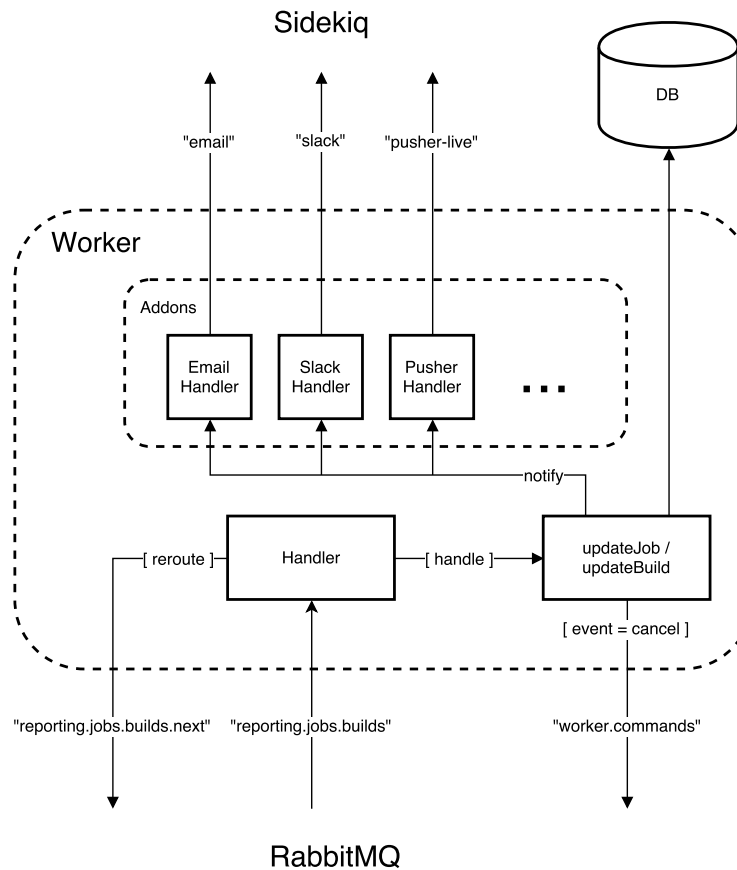


Figure 2.12: Worker

The addon services register themselves for different events and eventually pass a message to Travis Tasks. Tasks is further discussed in Section 2.4.6.

In this context rerouting means to pass the event to another queue to reduce the load on the Hub or handle it separately. The queue for rerouting is called “reporting.jobs.builds.next” as for AMQP or “hub” for Sidekiq (see Listing 2.9 line 32). Multiple reasons can cause rerouting mostly configured with environment variables. The running Heroku Dyno must have rerouting enabled. Additionally Redis is used to access a key to determine whether rerouting is enabled or not. If these conditions are met, rerouting can occur by owner or by percent as seen in Listing 2.9 at line 39.

2.4.5 Travis Logs

Travis Logs is a service that receives the output of the job execution from Worker (see Section 2.4.3). It assembles the streamed parts into a complete log and archives it to Amazon S3 after the job is finished.

The Logs service is composed of a separated database and five major components, performing four steps while processing a job. The database is separated from the

Listing 2.7: travis-hub/lib/travis/hub/service/update_job.rb

```

1 def run
2   exclusive do
3     validate
4     update_job
5     notify
6   end
7 end

```

Listing 2.8: travis-hub/lib/travis/hub/service/update_build.rb

```

1 def run
2   exclusive do
3     validate
4     update_jobs
5     notify
6   end
7 end

```

Listing 2.9: travis-hub/lib/travis/hub/support/reroute.rb

```

32 QUEUES = { amqp: 'builds.next', sidekiq: 'hub' }
33
34 def run
35   reroute || true if reroute?
36 end
37
38 def reroute?
39   dyno? and enabled? and (by_owner? or by_percent?)
40 end
41
42 def reroute
43   target = ENV['REROUTE_TARGET'] || :amqp # context.redis.get("#{name}_target")
44   queue = QUEUES[target.to_sym]
45   info "Routing #{type}:#{event} for id=#{object.id} to #{target}=#queue"
46   publisher = self.class.const_get(camelize(target)).new
47   publisher.publish(context, queue, [type, event].join(':'), payload)
48 end

```

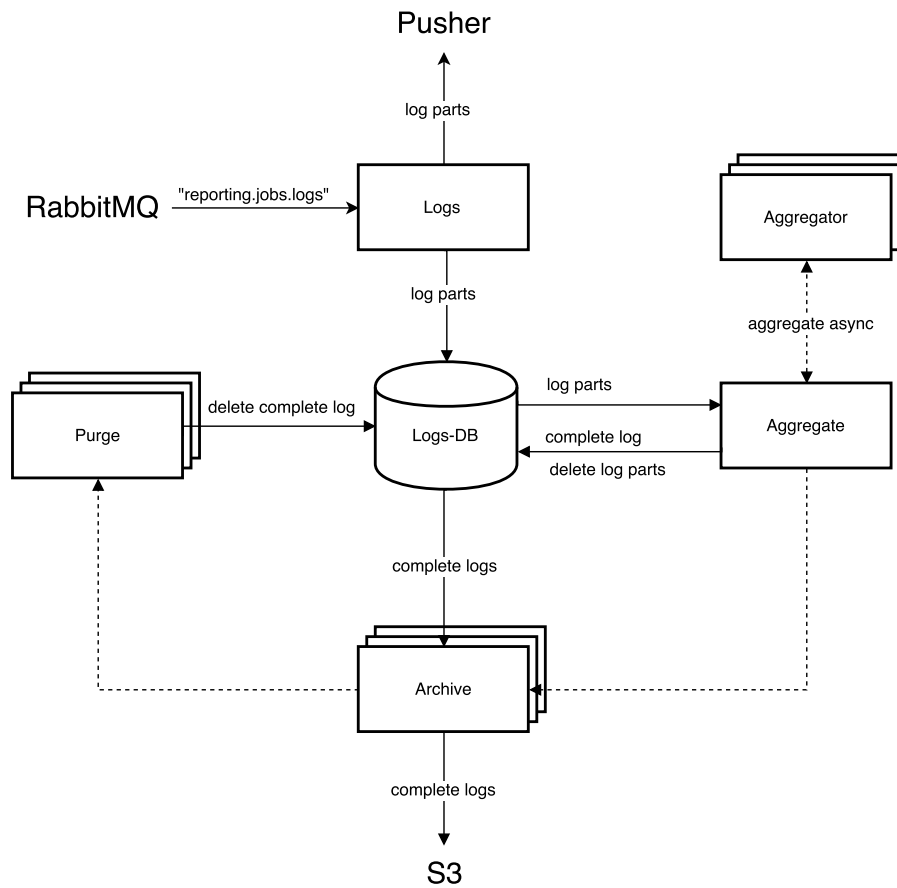


Figure 2.13: Logs Architecture

main Travis database as shown in Figure 2.4. The steps are *receive*, *aggregate*, *archive* and *purge*, displayed clockwise in Figure 2.13. The Logs component²¹ receives log parts, the Aggregator assists the Aggregate component and the other components are named after their step of execution. Aggregator, Archive and Purge represent multiple Sidekiq background processes, visualized as a stack of items in Figure 2.13.

Logs The Logs component receives streamed log parts from the Worker discussed in Section 2.4.3. It stores log parts in the database and sends out notifications via Pusher. The component consists of multiple threads subscribing to the “reporting.jobs.logs” RabbitMQ queue. The handler for events of that queue is the *ProcessLogParts* service shown in Listing 2.10. There is one log per job and every log part is associated with that log. A log ID is used, which equals the ID of the corresponding job. The first received part of each job creates a new log record in the database. Therefore each part checks if it must create a new log or find an existing one, since it is not

²¹The Logs component is a part of the Travis Logs service. In contrast to Section 2.4.4, subcomponents are capitalized in this Section.

guaranteed that all parts arrive in order of creation. After the corresponding log was found or created the part is stored in the database. If it is the last part, it is marked as *final* indicating to the Aggregator that the job is finished and all parts will be present in the database soon. Additionally each part saves its part number to ensure that parts can be sorted. Finally the Logs component notifies other services about the new log part using Pusher. For example, Travis Web receives this notification to update the displayed log, while a job is still running (see Figure 2.3). Services can subscribe to this Pusher channel, which is called `[private-]job-<jobID>`, using a web API. This API is a minor component of Travis Logs that uses a Sinatra application to allow creating webhooks for Pusher or directly editing the content of a log. Furthermore it keeps track of all subscriptions to this channel using Redis key value pairs that expire after six hours.

Listing 2.10: `travis-logs/lib/travis/logs/services/process_log_part.rb`

```
def run
  measure do
    find_or_create_log
    create_part
    notify
  end
end
```

Aggregate In this context *aggregation* means to concatenate all log parts of a job into one final log. The Aggregate component performs the aggregation, saves the result as the log content, and deletes the used log parts. This component permanently runs in an infinite loop. At first it identifies jobs, whose log parts are ready to be aggregated. These are either jobs including a log part that is marked as *final* or jobs with parts so old that they are forced to be aggregated. Even if a part is marked as *final* the aggregation is delayed by a preset time interval to guarantee that all parts are present in the database. By default the Aggregate component assembles the logs directly, but if the service is configured to aggregate the logs asynchronously, it queues this task using a Sidekiq queue. The Sidekiq background processes of the aggregator subscribe to that queue. Listing 2.11 contains the code that is used by the Aggregate and the Aggregator components. They concatenate the parts using a SQL statement, delete all log parts that were concatenated, and queue the log archiving to the “archive” queue. The deletion of the log is done in the vacuum method.

Archive After aggregation the complete log is present in the database. Although all included parts were removed during aggregation the content of all assembled logs adds up to a massive amount of data. To reduce stored data, the Archive component saves the logs to Amazon S3 so they can be deleted by the Purge component. Archive is a collection of Sidekiq background processes just like Aggregator and

Listing 2.11: travis-logs/lib/travis/logs/services/aggregate_logs.rb

```
def aggregate_log(log_id)
  transaction do
    aggregate(log_id)
    vacuum(log_id) unless log_empty?(log_id)
  end
  queue_archiving(log_id)
  Travis.logger.debug "action=aggregate log_id=#{log_id} result=successful"
rescue => e
  Travis::Exceptions.handle(e)
end
```

Purge. Archive responds to events from the “archive” queue and executes the code shown in Listing 2.12.

If no log record exists for the given job, archive does nothing. While performing any successive steps, the log gets marked as being in *archiving* state. In case the log exists and is not empty, it can be stored in the Amazon cloud as a plain text file. Afterwards it is verified that the archived file and the log in the database are of equal size. This is confirmed in the database by marking the log as *archive_verified* and setting the archiving time. Eventually an event to the “purge_log” Sidekiq queue initiates the deletion of the log from the database.

Listing 2.12: travis-logs/lib/travis/logs/services/archive_log.rb

```
def run
  return unless fetch
  mark_as_archiving
  return if content_blank?
  store
  verify
  confirm
  Travis.logger.debug "action=archive [...] result=successful"
  queue_purge
  investigate if investigation_enabled?
ensure
  mark_as_archiving(false)
end
```

Purge The Purge component performs the last processing step for a log, after it was archived to Amazon S3. A collection of Sidekiq processes remove the content of archived logs from the database. The meta data is not deleted, though. Different possible errors during the last steps are covered before purging the log content. Empty log content or empty S3 files result in error messages that must be handled manually. Furthermore, if the length of the archived and the database log does not

match, the archiving process is queued again. The actual purging sets the purge time and removes the log content as seen in Listing 2.13. Meta data such as archiving time persists in the logs database.

Listing 2.13: travis-logs/lib/travis/logs/helpers/database.rb

```
def purge(log_id)
  @db[:logs].where(id: log_id).update(purged_at: Time.now.utc, content: nil)
end
```

2.4.6 Travis Tasks

Travis Tasks integrates 3rd party services into Travis. An external service is represented by a *Task* that sends out notifications and a *Handler* to acquire information and start the Task. Travis Hub runs the Handler when an event occurs the Handler subscribed to. Since Hub has a database connection, it gathers all data for the Task to run before notifying Travis Tasks (see Section 2.4.4). A Task is executed without database connection by Travis Tasks and communicates with an addon service like mail or slack.

Hub’s responsibility is the delivery of events and not the execution of Tasks. Therefore each Handler only passes the event to Travis Tasks supplementing information from the database. A Task is allowed to run longer than the Handler, because it is run by Tasks and not by Hub. The Pusher Task is an exception, since it is not executed by Tasks. *Travis Live* receives and processes events from Sidekiq’s “pusher-live” queue and executes the Pusher Task.

Figure 2.14 shows an excerpt of Travis Tasks. The three Tasks correspond to three Handlers, two of which are shown in Figure 2.12. Each Task listens for events on an unique queue. The GitHub Task uses the “github_status” queue and is responsible for displaying test results in pull request on GitHub. This represents the fourth step of the *Pull Request Build Flow* from Figure 2.1. The Mail Task sends emails and the Slack Task writes comments in multiple Slack channels. Listing 2.14 shows a method of the Slack Task, which is executed once per target channel. Additionally this Task checks the channel for format errors and fills a template message with data, such as build result and repository slug.

2.5 Evaluation

In this section we evaluate the consequences of Travis’ architecture in terms of modularity, extensibility, and scalability.

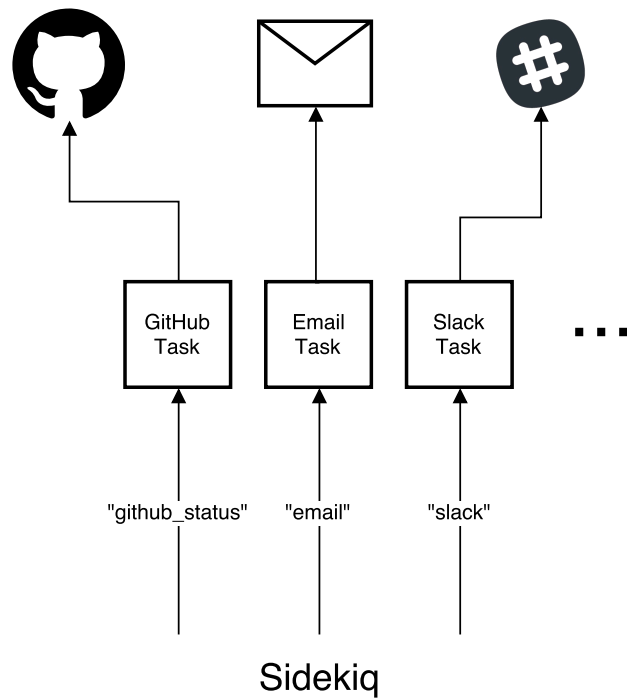


Figure 2.14: Tasks

Listing 2.14: `travis-tasks/lib/travis/addons/slack/task.rb`

```
def send_message(target, timeout)
  url, channel = parse(target)
  http.post(url) do |request|
    request.options.timeout = timeout
    request.body = MultiJson.encode(message(channel))
  end
end
```

Modularity Travis' architectural approach is based on reductionism rather than holism. Seeing the whole complex system as no more than the sum of its parts accomplishes a modular structure, where components can easily be removed or added [25, p. 312 ff.]. Most modules like Listener or Hub use message broker software or asynchronous background processes for communication, hence are loosely coupled. Currently the central database and shared helper repositories are invariant parts of the system. However, the shared repository *travis-core* originates from the historical implementation and is going to be removed.

Extensibility In most cases it is simple to extend Travis, because of its modularity. A user can add custom scripts to their configuration file that allow for versatile additional behavior to the test builds. Although the scripts are not as powerful as Jenkins' plugins, their use case is completely different. While plugins directly extent stages of the build pipeline or add custom ones, Travis' user scripts contain custom commands that are hooked into the pipeline at preset places like `before_install`, `after_success` or `before_deploy`. In fact one can overwrite the default testing script, but not manipulate it [46, Section Customizing the Installation Step]. Since Travis is open source, functionality can be added to the system itself. Most Travis repositories have a *Contributing* section explaining how to propose a new feature or bugfix.²² On the one hand, the actual change most often affects only a handful of modules. For periodic builds, which represent a completely new build flow, just one module was changed (chapter 3). On the other hand, this kind of distributed-system development requires some familiarity with supporting services, libraries, and open source development. As we worked on implementing the periodic so called *Cron Jobs* (chapter 3), we sometimes struggled to understand the existing codebase, due to the years of experience that were put into the system.

Scalability As mentioned in Section 2.4, all modules are deployed on Heroku. Heroku's process model provides the means to easily scale up and down. Therefore the capacity of nearly all modules can instantly be increased solely limited by upkeep cost. The Heroku Website states that 'scaling your app out horizontally is as simple as dragging a slider'²³ Furthermore the Hub module is able to reroute requests to another Hub module, which lifts additional load from a single Hub. So in terms of scalability of the existing structures, Travis limits are far from being reached. Nevertheless, problems arise when trying to scale the system by adding support for other version control systems or repository hosting services. MagnumCI is an example of a cloud based CI service that supports the repository hosting services GitHub, GitLab, and Bitbucket, but to integrate Bitbucket into Travis, enormous efforts would be needed. Currently all members are verified by their GitHub account, multiple modules gather additional information like commit messages directly from GitHub, the build script downloads tested repositories from GitHub, the Listener

²²<https://github.com/travis-ci/travis-api/blob/master/README.md>
(visited on 2016-06-15).

²³<https://www.heroku.com/pricing> (visited on 2016-06-15).

depends on a predefined GitHub Webhook, and the support for GitHub pull requests is directly integrated into Travis. Apart from changing nearly all modules to support multiple repository hosting services, the database would need to distinguish between GitHub content and content from other sources.

2.6 Future Work and Conclusion

In this chapter, we gave an overview of Travis' architecture and functionality. Since Travis is open source, the community frequently requests new features and improvements. Travis decides whether they are worth the effort. There currently are multiple features in progress.

Tackling the problem of software that runs out of date or becomes incompatible with their dependencies without a change in the software itself, an alternative build flow to the process illustrated in Figure 2.1 is required. Builds could be triggered periodically (chapter 3) or on every change of the dependencies (chapter 4). The former just needs changes to Travis API, while the latter depends on substantial adjustments to multiple modules and possibly a completely new module. What all solutions for that problem – beyond the two mentioned – have in common, is that they modify the existing build pipeline by adding new starting points aside from pushing changes to GitHub.

Moreover, the way in which results are presented to the customer can be improved. Whereas the present approach shows individual build results, it is difficult for the user to gather information that depends on multiple data sets. Therefore a central access to combined test results, where information is presented in a visual and intuitive way, would highly increase the benefit of using a continuous integration service in the first place (chapter 5).

The extent of modularity is based on the change from a single repository in 2011 to many microservices [35]. Looking at the enormous architectural changes during that time period, Travis' goal of modifying the system towards modularity, extensibility, and scalability remains. In the future, this process continues with the planned removal of *travis-core*.

In conclusion, Travis provides a structure that can easily grow and be extended, not only by their own employees but also by the open source community. Especially the users can customize their experience with Travis, using configuration files, own build scripts, and the settings on the web page.

3 Interfaces for Expressing Periodic Task Schedules

Periodic task schedules are required when performing tasks regularly. Currently, Crontabs are the predominant mean of expressing periodic task schedules in IT systems. Expressing periodic schedules with Crontabs is difficult, overly specific, and tainted with expressiveness issues. We propose and evaluate different approaches of expressing periodic task schedules, trying to avoid the issues of Crontabs. Our implementation realizes point-vector periodicity, the most suitable solution we found. This work provides an overview about properties of different periodic schedules and aims for helping the reader choose the most appropriate solution for their use case.

3.1 Introduction

When performing tasks on a regular basis, periodic task schedules are required to express *when* a task has to be executed. In IT systems, the Cron job scheduler is the most common solution for recurringly performing tasks. The Cron daemon uses Crontabs as interface for expressing periodic schedules, stating when a task has to be executed.

The problem with Crontabs is, that they only provide one, fixed scheme for expressing periodic schedules. This scheme is hard to understand, provides only one degree of precision, and has expressiveness issues.

In this chapter we focus on problems connected with Crontabs in the context of continuous integration systems [5, Continuous Integration, p. 49–50] and examine different approaches for expressing periodic task schedules. We evaluate the approaches in terms of understandability, precision, and expressiveness and propose a terminology. Our implementation of the most intelligible solution we found takes place in context of the continuous integration service Travis CI and enables periodic testing of software projects.

3.2 Context

Initially, building, testing, and integrating was done whenever the developers had time to do so or thought it to be reasonable. There was no continuity or periodic schedule for performing these steps. To avoid large change sets that might be hard

to integrate or could break large parts of the systems at once,¹ continuous integration was introduced. At first, continuous integration was done only time-based. This was due to limited resources. Building, testing, and integrating took a vast amount of time which forced the developers to perform this step at night, when there was no one working on the system (known as “nightly builds”). As computing power improved, continuous integration could be performed more often. In combination with version control systems, building, testing, and integrating can be done subsequently to every commit. This kind of event-driven continuous integration is used by Travis CI. Unfortunately, there may be external dependencies or resources that are out of the version control systems scope, like databases, media files, or other software that is used by the project but that is not under accessible version control. These external resources may change without the version control system noticing. While this is not a problem for actively developed projects because these projects are frequently changed and therefore built and tested, it tends to be problematic for idling projects. Building periodically (time-driven) in addition to building after each commit (event-driven) is a generic way of coping with this kind of problem. Depending on how often external resources change, the interval of periodic building can be chosen larger or smaller. This leads us to the problem of how to express intervals and periodicity.

In computer science, *Crontabs* (cron tables) are a common way of expressing schedules for periodic tasks. Crontabs are used with the Cron daemon, a time-based task scheduler that usually runs on Unix-like systems. The standard [44] way of specifying recurring events in a crontab entry is to set values for the fields “minute”, “hour”, “day of month”, “month”, and “day of week” whereas the values can either be a do not care symbol (*), an element, or a list of elements (comma separated). An element can either be a number (see below) or two numbers separated by a hyphen (–) which specifies an inclusive range. Permitted numbers for elements are the following:

minute 0 to 59 (the minutes of an hour)

hour 0 to 23 (the hours of a day)

day of month 1 to 31 (the days of a month)

month 1 to 12 (the months of a year)

day of week 0 to 6 (the days of a week as numbers starting at (0 =) Sunday)

The task associated with the crontab entry runs whenever the current time matches the entry’s pattern which is the case when the time part (“minute” and “hour”) and the date part (“day of month”, “month”, and “day of week”) are matched by the current time. The time part is matched whenever the current minute matches the “minute” field of the pattern and the current hour matches the “hour” field of the pattern. To find out if the date part is matched we have to distinguish 4 cases:

¹Defects in the code can only be noticed, if there are exhaustive tests. At least one point in the infection chain of the defect has to be covered by a test to avoid missing the defect. Otherwise the user will observe a failure which has to be avoided.

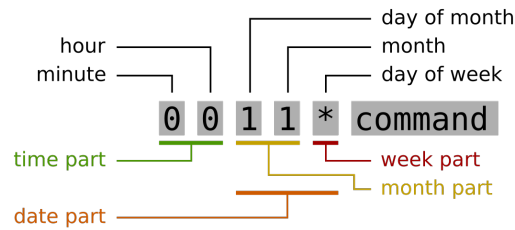


Figure 3.1: Naming of structural parts of a crontab entry in this paper

1. The fields “day of month”, “month”, and “day of week” are all unspecified (filled with the do not care symbol, *). In this case all days are matching the pattern.
2. The field “day of month” and/or the field “month” are specified but “day of week” is not. In this case a day matches the pattern if and only if it matches both the “day of month” and the “month” field.
3. Both “day of month” and “month” are unspecified but “day of week” is. In this case a day matches the pattern if and only if it matches the “day of week” field.
4. The field “day of month” and/or the field “month” are specified and “day of week” is also. In this case a day matches the pattern if it matches the “day of month” and the “month” field and/or if it matches the “day of week” field.

A value matches a pattern field, if the field’s content is either a do not care symbol (*), an element that is matched by the value, or a list of elements that contains one element that is matched by the value. An element is matched by a value if either the element is a number and the number is equal to the value, or the element is an inclusive range which includes the value [44].

3.3 Problem

Despite the fact that Crontabs are a popular way of expressing periodic time schedules in terms of computer science, they also have crucial disadvantages.

Understandability First of all the format of Crontabs is hard to map on natural language and hence not intuitive. Since Crontabs are basically a succession of numbers followed by a command, the user has to *know* what each number means. The fact that Crontabs are not self-explanatory is primarily a difficulty for new users. But even if the user is acquainted with the meaning of each field there are still difficulties in translating the Crontabs time format to natural language. A crontab

"0 0 1 1 * command" is read as "when the minute is zero, the hour is zero, the day of month is one, and the month is one". Firstly it is unclear, why "minute", "hour", and "day of week" start at zero while "day of month" and "month" start at one, and secondly this is a very complicated way of expressing "yearly" which leads to the second disadvantage of Crontabs.

Overspecification Whenever something has to happen once in a period of time, all smaller units of time must be specified as well. As shown in the example above, when a command has to be executed once a year, the month, the day of month, the hour, and the minute have to be set too. There are implementations like "Vixie Cron" by Paul Vixie [48, 49] that try to cope with the problem of overspecification by adding keywords that can be used instead of the five fields used in the standard implementation. "Once a year" can then be expressed as "@yearly command" which translates to "0 0 1 1 * command" [49]. The problem with this notation is that it only affects the way the user deals with Crontabs, not the way crontab entries are processed by the computer. While the user only specifies that something has to happen yearly while being unspecific about when exactly it has to happen, the computer interprets "@yearly" as "exactly at the beginning of the year". While this is inconsiderable for one cron job many cron jobs with this kind of specification will cause bursts at the beginning of common time intervals (e.g. years or months).

Expressiveness In our opinion the main disadvantage of Crontabs is that the expressiveness is not suitable for some common scenarios. One problem is that expressing fixed intervals of more than one unit is hard or even impossible. While it is easy to state that something should happen once in a time interval corresponding to a field of a crontab entry (i.e. once a minute, hour, day, week, or month) it is hard to express that something should happen once in a time interval that is a multiple of an interval corresponding to a crontab field. For example, expressing "execute command once a day" is easily expressed by "0 0 * * * command" but a minor modification like "execute command once in *two* days" is difficult to express.

Another problem is that at least the standard implementation of Crontabs is limited to the five fields "minute", "hour", "day of month", "month", and "day of week". Holidays, phases of the moon, the weather, and other interesting measures like how often a day of the week appears this month are not taken into account. A use case of these measures is the opening hours of some public authorities which are on duty for example every fifth Wednesday a month. If there are only four Wednesdays a month there has to be an alternative opening time like the fourth Wednesday or first Wednesday of the next month.

Also, definitions of dates that are defined relatively to other dates (like Easter) are not to be expressed with crontab entries. For example, the date of Easter is defined as the first Sunday after the first full moon in spring. Despite the fact that moon phases are not supported in cron tabs, it is also not possible to state something to be the n-th

occurrence of x before or after y . Depending on the domain this may or may not be a problem.

To discover more time specifications that are not to be expressed with Crontabs we have to take a closer look on how the fields of a cron tab entry are connected. As described above there is a part for time specification and a part for date specification in a crontab entry. The fields of the time part (“minute” and “hour”) are simply connected by a logical conjunction which means that the current “minute” has to match the “minute” field of the pattern *and* the current “hour” has to match the “hour” field of the pattern. The connection between the fields of the date part (“day of month”, “month”, and “day of week”) is more complex. While “day of month” and “month” are also connected by a logical conjunction the connection between these two fields and “day of week” is a logical *disjunction* as long as both “day of week” and at least one of “day of month” and “month” are specified. If the week part or the month part is unspecified (*) then it is a conjunctive relationship. Expressed in natural language this means that “day of month” *and* “month” have to match the pattern *or* “day of week” has to match the pattern if both parts are specified.² If only the month part (“day of month” and/or “month”) is specified and the week part (“day of week”) is * only the month part has to match and vice versa: if only the week part is specified only the week part has to match. The logical disjunction deprives us of the possibility to express that something has to happen if and only if all three fields are matched. Expressing “every Sunday in June” or “if the first day of the month is a Monday” is impossible due to the use of a logical disjunction instead of a conjunction. It may seem that it comes in handy to be able to specify patterns like “every Monday and also every first day of the month” (“0 0 1 * 1 command”) with one crontab entry but while this can also be expressed with two Crontabs entries (“0 0 1 * * command” and “0 0 * * 1 command”) a conjunctive relationship between “day of month”, “month”, and “day of week” can not be expressed, neither with one nor with many crontab entries. This is because crontab entries are disjunctively connected among each other. A command is executed if one crontab entry is matched by the current time, not all crontab entries that contain this command have to be matched.

3.4 Approach

We found two common approaches for expressing periodic task schedules. The first is “point-vector periodicity”. Using this approach, periodic schedules are specified with a starting point in time and a duration. A given point in time belongs to the periodic set if it is reachable by adding a multiple of the duration to the starting point. This approach perfectly matches terms like “starting now every two days” which is a

²The disjunctive relationship is between week part and month part may be for convenience reasons for experienced users, assuming that these disjunctions (e.g. “every Monday and every first day of the month”) are a common use case, while conjunctions (e.g. “if the first of the month is a Monday”) are highly improbable and can hence be ignored.

common way of expressing periodicity in natural language. The second approach is “conditional periodicity”. Crontabs are, among others, part of the second category. When using this approach, recurring points in time are specified by conditionals. A point in time belongs to the set of recurring events if and only if the conditionals can be evaluated to true. In the following we will describe how these two main concepts can be used.

3.4.1 Point-Vector Periodicity

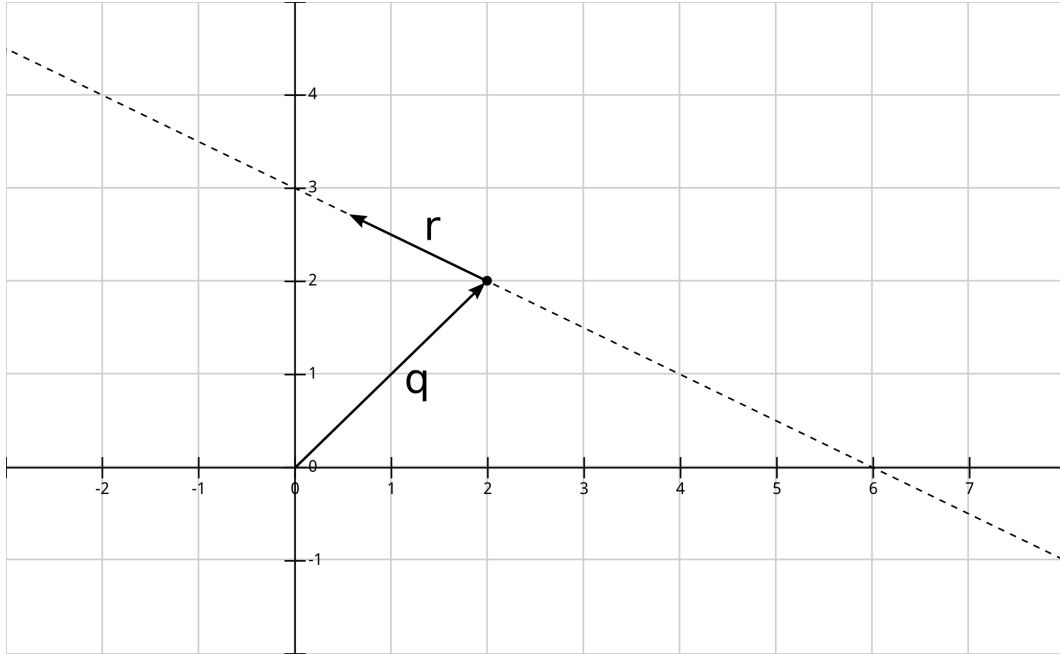


Figure 3.2: Linear function defined by a point (position vector) \vec{p} and a vector \vec{q}

Definition In math, linear function can be defined by a starting *point* and a *vector* using the following formula, whereas \vec{p} is the position vector of an arbitrary point on the graph of the linear function, \vec{q} is the position vector of the starting point and \vec{r} is the direction vector:

$$\vec{p} = \vec{q} + s \cdot \vec{r} \quad p, q, r \in \mathbb{R}^n \quad s \in \mathbb{R}$$

Or, as a concrete example for two-dimensional linear functions (Figure 3.2):

$$\begin{pmatrix} p_x \\ p_y \end{pmatrix} = \begin{pmatrix} q_x \\ q_y \end{pmatrix} + s \cdot \begin{pmatrix} r_x \\ r_y \end{pmatrix} \quad p_x, p_y, q_x, q_y, r_x, r_y, s \in \mathbb{R}$$

We propose a similar approach for specifying periodic time schedules, were a periodic set of points \vec{p} in time is declared by a starting point in time \vec{q} and a duration \vec{r} .

$$\vec{p} = \vec{q} + s \cdot \vec{r} \quad s \in \mathbb{Z}$$

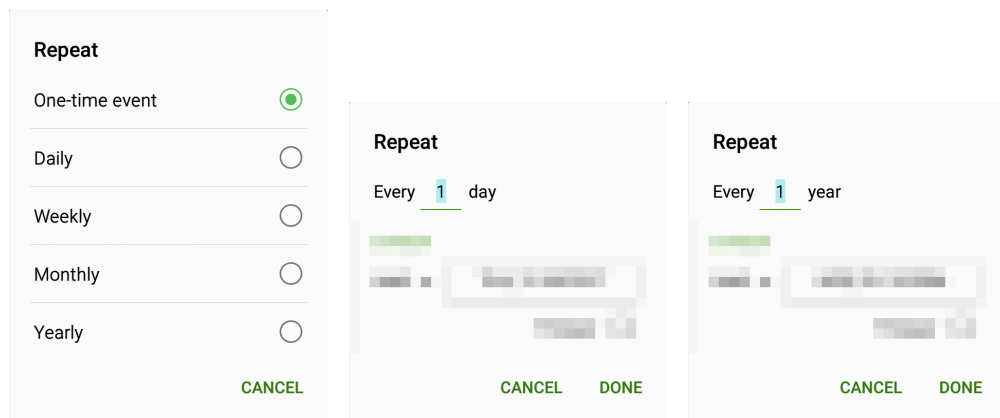
Please notice that s is an integer in the latter formula, because if s would be a real number every point \vec{p} in time would match. Since time is scalar quantity, this formula can also be considered an arithmetic progression, which leads to the following formula:

$$t_i = t_0 + i \cdot \Delta t \quad i \in \mathbb{Z}$$

For encoding the starting point t_0 in time and the duration Δt we can either use numbers representing units of time, e.g. seconds or milliseconds, since a fixed point in time (like Unix timestamps) or we could use any other date and time format as long as arithmetic operations (at least addition) are defined on it. Consider this example using RFC3339 [30] notation:

$$t_i = 2016-05-11T11:08:12+02:00 + i \cdot 0000-00-03T12:00:00Z$$

This defines a set of periodic events starting at 11 May 2016 11:08:12 in UTC+2 (Central European Summer Time) and recurs every 3 days 12 hours. The example shows that using point-vector periodicity is intuitive since it is easy to map on natural language. This may be the reason why this kind of specification (or variations therefrom) is often used in end-user calendar software (Figure 3.3).



(a) Selecting unit for interval Δt (b) Selecting the numerical value for Δt when unit is "days" (c) Selecting the numerical value for Δt when unit is "years"

Figure 3.3: Creating recurring events with Samsung's SPlanner

Simplification Complexity reduction can be achieved by only allowing specific values for the duration Δt . Calendar software usually offers to set durations that are multiples of common units like day, weeks, or months, as shown in Figure 3.3. This makes sense e.g. for meetings that take place every second week whereas in context of continuous integration it is only of minor importance whether a project is tested every week or only every two weeks. Therefore the domain of Δt can be condensed to a choice of magnitude, e.g. by only allowing the values “one day”, “one week”, and “one month”.

Another mean of reducing complexity for the user is to avoid setting the starting point t_0 manually³ because when dealing with external dependencies it is more relevant how often a project is rebuild and not when exactly. With respect to this the current time (“now”) can be taken as default value. Furthermore, the domain of i can be reduced. For example by limiting i 's domain to the set of natural numbers (\mathbb{N}_0) we can be sure that t_0 is the first event in the set. All times before t_0 are not to be taken into account. This makes it easier for the users to consider t_0 to be the starting point of the series of events.

Limitations When using point-vector periodicity only fixed intervals can be expressed due to the fact that Δt is fixed (Figure 3.4). This leads to equally sized intervals (of size Δt) which is on the one hand often the desired behavior but on the other hand this makes it impossible to express periodic sets with varyingly sized intervals.

Comparison with Crontabs As mentioned above, point-vector periodicity is easy to map on natural language and therefore, in contrast to Crontabs, easily understandable. A problem concerning understandability that occurs when using point-vector periodicity with arbitrary duration (Δt) for longer time is that it gets harder to find out if a given point in time t_i is part of the periodic set due to the fact that i has to grow with time. To cope with this problem, the domain of Δt can be reduced to values that allow calculations of t_i by only manipulating specific parts of t_0 . For example, “1 day” as value for Δt allows optimization by only comparing every field with lower dimension than the day field (i.e. hours, minutes, seconds, and possibly milliseconds) of t_0 with t_n . If the fields match we can be certain that t_n is part of the set defined by t_0 and Δt . Unfortunately this approach limits the expressiveness crucially. An alternative approach is using a recursive definition of the arithmetic progression rather than the explicit form. This changes the formulas in use as follows:

$$t_i = t_0 + i \cdot \Delta t, \quad i \in \mathbb{Z} \quad (\text{explicit formula})$$

becomes

$$t_{i+1} = t_i + \Delta t, \quad i \in \mathbb{Z} \quad (\text{recursive formula})$$

³Selecting points in time is a complicated job because date and time specification requires many components that have to be taken into account.

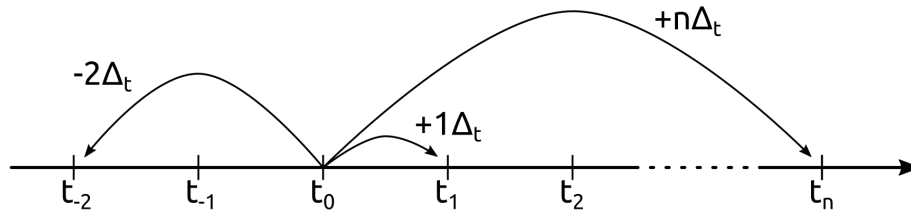


Figure 3.4: Expansion of point-vector periodicity (using the explicit formula) by adding $i\Delta t$ for varying $i \in \mathbb{Z}$ and fixed Δt

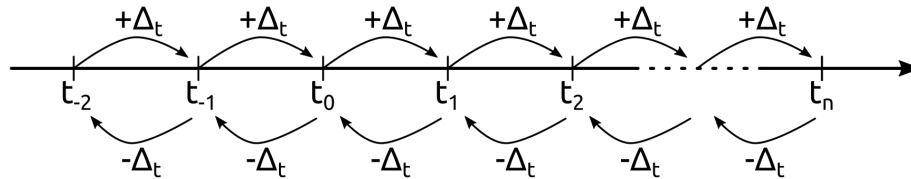


Figure 3.5: Expansion of point-vector periodicity (using the recursive formula) by adding or subtracting Δt to or from a given t_i

When using the recursive formula, we need to store t_i and update it every time, due to the fact that every t_{i+1} is only to be calculated if t_i is known, as shown in Figure 3.5.

The problem of overspecification is not inherently solved by point-vector periodicity. The degree of specification depends on how precise the date and time format in use is. Most programming languages implement date and time formats with precision up to milliseconds which leads to enormous overspecification. Concerning expressiveness, point-vector periodicity is only able to avoid one of the disadvantages described in section 3.3, while even creating some new. Point-vector periodicity allows creating equally sized intervals of multiples of common time units, e.g. 2 days.

3.4.2 Conditional Periodicity

Definition Conditional periodicity provides logical expressions rather than arithmetic formulas to express periodic schedules. A given point in time is part of the set defined by a logical expression if and only if the logical expression can be evaluated to true for the given point in time. Logical expressions are created by linking predicates using logical operations. While logical operators, like conjunction, disjunction, and negation, are well known and universally usable, it is unclear how to express predicates in terms of points in time and periodic expressions. We need to access specific properties of a point in time to use in predicates. For example, a predicate expressing that a given point in time t has to be in July can be expressed in a functional

way by:

$$month(t) = 7$$

Also, we could use an object oriented-notation, like Python's datetime module [15], and write:

$$t.month == 7$$

In this chapter we will use the functional notation because it abstracts from the actual implementation. When using a function for accessing properties of a point in time t , t can be anything, even an integer representing seconds since a fixed point in time. By using the object-oriented notation we are accessing a property of an object while not knowing if the property exists⁴

Now that predicates and junctions between them are known, the way of expressing a set of periodic events has to be considered. While there are other notations in use, we make use of the mathematical way of denoting sets:

$$S = \{ t \mid month(t) = 7 \}$$

This set contains every point in time that is in July, regardless of its year, day, hour, minute, second or any other measure. Expressing multiple conditions can be done as follows:

$$S = \{ t \mid (month(t) = 7 \vee month(t) = 1) \wedge day(t) = 1 \}$$

Expressing Crontabs The notation defined above can be used to specify an isomorphism for Crontabs. This shows, that Crontabs are just an instance of conditional periodicity. Using the set notation, a crontab⁵

$$\langle m \rangle \langle H \rangle \langle d \rangle \langle M \rangle \langle E \rangle \langle \text{command} \rangle$$

can be expressed as

$$S = \{ t \mid \begin{aligned} &minute(t) = \langle m \rangle \\ &\wedge hour(t) = \langle H \rangle \\ &\wedge ((month(t) = \langle M \rangle \wedge day(t) = \langle d \rangle) \vee weekday(t) = \langle E \rangle) \end{aligned} \}$$

⁴Assuming `t.month` is an attribute and not a getter method.

⁵In compliance with the Unicode Locale Data Markup Language [17], the following symbols are used:

- `<m>` for minutes with *minute* as corresponding function name
- `<H>` for hours (24-hours format) with *hour* as corresponding function name
- `<d>` for days of the month with *day* as corresponding function name
- `<M>` for months with *month* as corresponding function name
- `<E>` for days of the week with *weekday* as corresponding function name

While this works fine if the fields only contain single values, other forms of predicates have to be used if the fields contain lists of values or ranges. The normal form for single value fields is:

$$attribute(t) = value$$

Lists of values can be expressed with:

$$attribute(t) \in \{value_1, value_2, \dots, value_n\}$$

Ranges of values can be expressed with:

$$attribute(t) \in [form, to]$$

An asterisk, the do not care symbol in terms of Crontabs, is expressed by simply omitting the condition for this attribute. With respect to these replacement rules, the set expressing the crontab entry can be rephrased using only set notation for the attribute conditions while avoiding conditions of the form $attribute(t) = value$.

$$S = \{ t \mid \begin{aligned} &minute(t) \in \langle m \rangle \\ &\wedge hour(t) \in \langle H \rangle \\ &\wedge ((month(t) \in \langle M \rangle \wedge day(t) \in \langle d \rangle) \vee weekday(t) \in \langle E \rangle) \} \end{aligned}$$

In this set definition, $\langle m \rangle$, $\langle H \rangle$, $\langle d \rangle$, $\langle M \rangle$, and $\langle E \rangle$ always have to be sets. If the corresponding crontab fields contains a single value, the set representing the field also contains only a single value. For lists of values, every value in the list is in the set. Using ranges, the set is defined by an equivalent range. Do not care symbols are still represented by omitting the condition. By applying these rules we can translate crontab entries to the more generic form of conditional periodicity introduced in this section.

Improving Crontabs Now that we are able to express crontab entries in a generic way, we can make improvements and try to solve the identified issues. A simple but effective advance that solves some of the expressiveness issues is replacing the logical disjunction by a conjunction. This leads to the following set definition:

$$S = \{ t \mid \begin{aligned} &minute(t) \in \langle m \rangle \\ &\wedge hour(t) \in \langle H \rangle \\ &\wedge month(t) \in \langle M \rangle \\ &\wedge day(t) \in \langle d \rangle \\ &\wedge weekday(t) \in \langle E \rangle \} \end{aligned}$$

With this improvement, it is possible to express combinations between weekday and day of the month by specifying predicates for both. Expressing patterns like “if the first day of the month is a Monday” was not possible with Crontabs due to the fact that specifying both weekday and day of the month would mean “whenever at least one of the two predicates are matched” or more concrete: “every first day of the

month and every Monday". The improvement allows expressing "if the first day of the month is a Monday" appropriately. Moreover, this does not deprive us of the possibility to express "every first day of the month and every Monday". We just have to specify two sets of the predefined form.

Another problem solved is correlation between weekdays and months. In section 3.3 we figured out that events that have to take place on specific weekdays in specific months are not to be expressed with crontab entries due to the disjunctive connection between week part and month part. With the conjunctive connection in our improvement, this is possible.

While the advance made above only makes use of the fields used by Crontabs, other information can be included as well, as long as they can be represented by a function. The functions used do not necessarily have to return a numeric value. For example, holidays could be queried by a function $isHoliday(t)$ which returns a boolean value.

Limitations While the proposed general form of conditional periodicity is able to solve most of the expressiveness problems Crontabs have, there are some problems that remain unsolved using this approach. Firstly, it is still unclear how to express equi-sized intervals of arbitrary length like in point-vector periodicity. This issue will be addressed in the next paragraph when dealing with hybrid forms of conditional and point-vector periodicity. Secondly, understandability is only given when assuming the user to have knowledge of formal logic. Conditional periodicity in general, as well as Crontabs in particular, does not map well on natural language. While conditional periodicity can be a powerful tool in terms of computer science and mathematics, it is not suitable for end-user applications due to the fact that it requires knowledge about formal logic. Thirdly, the level of detail is determined by the domain of t in a set $S = \{t | \dots\}$. If t represents time with a precision up to milliseconds, the specification has to be accordingly specific. For crontab entries, we assume that the granularity of t is not smaller than minutes. Unfortunately, the overspecification problem is not influenced by the improvements made. If specifying periodicity with a coarse granularity (e.g. "every day"), every smaller entity of time has to be specified as well (e.g. "every day at 18:00:00.000"). A solution is proposed in section 3.4.5 where we introduce vague events.

3.4.3 Hybrid Forms

Since the inability of expressing equi-sized intervals of arbitrary length is the only weakness in terms of expressiveness conditional periodicity has and point-vector periodicity solves exactly this problem, it seems natural to combine these two forms of expressing periodicity. We propose using the arithmetic progression that specifies point-vector periodicity as an additional predicate for the conditional periodicity that

is connected to the other predicates by a conjunction. Thus, point-vector periodicity is built-in into conditional periodicity⁶

$$S = \{ t \mid t = t_0 + i \cdot \Delta t \wedge i \in \mathbb{Z} \wedge \dots \}$$

For reasons of clarity and comprehensibility, conditional periodicity and point-vector periodicity can be defined as two different sets. These sets are then joined together by an intersection.

$$\begin{aligned} S_{pv} &= \{ t \mid t = t_0 + i \cdot \Delta t \wedge i \in \mathbb{Z} \} \\ S_{cond} &= \{ t \mid \dots \} \\ S &= S_{pv} \cap S_{cond} \end{aligned}$$

This procedure is equivalent to the one described above because the intersection between set A (S_A) and set B (S_B) is by definition a set that contains a value if and only if the very value is contained in both sets – in S_A *and* in S_B . Therefore:

$$\begin{aligned} S &= S_A \cap S_B \\ \rightarrow S &= \{ x \mid \langle \text{conditions of } S_A \rangle \} \cap \{ x \mid \langle \text{conditions of } S_B \rangle \} \\ \rightarrow S &= \{ x \mid \langle \text{conditions of } S_A \rangle \wedge \langle \text{conditions of } S_B \rangle \} \end{aligned}$$

3.4.4 Time Range Limitation

By now, sets of periodic events are reaching infinitely into the past and the future. In the context of point-vector periodicity we tried to limit at least one direction of infinity by specifying the domain of i to be \mathbb{N}_0 rather than \mathbb{Z} . While this avoids t_i to be less than t_0 , t_i can still get infinitely big. In terms of conditional periodicity, more advanced limitations in time range can be done. Ordering relational operators, like $<$, \leq , $>$, and \geq , enable precise delimitation of t 's range. Predicates using these operators for constraining the range of t are simply connected with the other predicates in use via a conjunction.

An example using RFC3999 for denoting time:

$$\begin{aligned} S &= \{ t \mid t \geq 2016-05-11T11:00:00+02:00 \\ &\quad \wedge t < 2016-06-11T11:00:00+02:00 \\ &\quad \wedge \dots \} \end{aligned}$$

3.4.5 Vague Events

Another issue with Crontabs we have shown in section 3.3 was overspecification. The version of Crontabs we refer to is always specific to the minute. For use cases

⁶The three dots in the equation (...) are a placeholder for the predicates of the conditional periodicity.

where there is no need for such a high precision we present three approaches to address this problem:

1. coarse domain
2. allowed deviation
3. time frames

These solutions offer periodic time schedules that are widely defined and leave therefore room for other concerns, like load balancing or priority scheduling.

Coarse Domain Vague events can be achieved by defining the points in time t that are part of the periodic set S with a domain D that is appropriately coarse-grained. If a deviation of maximum an hour around t is acceptable, the granularity of D can be limited to hours as smallest unit of measurement. Minutes, Seconds, and every smaller unit do not have to be taken into account. With this approach either a component based format (storing years, months, days, etc. separately) can be used and clamped after (for our example) the hour component, or a format that counts time units since a fixed point in time (e.g. begin of 1970) can be used whereas the time unit counted has to be hours for the example. Figure 3.6 illustrates the example.

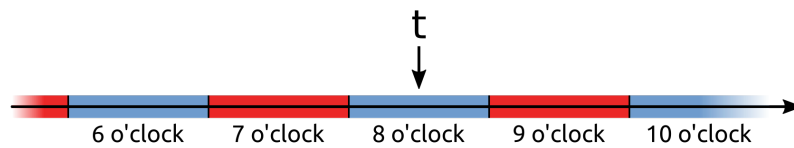


Figure 3.6: Timeline showing the coarse domain approach with a granularity of one hour. Every red or blue bar represents one hour and bars can only be selected as a whole.

Allowed Deviation Another way of expressing a vague periodic schedule is specifying the allowed deviation explicitly. Consider the following periodic set described by conditional periodicity with the domain D limited to minutes as smallest considered unit⁷:

$$S = \{ t \mid t \in D \wedge hour(t) \bmod 6 = 0 \wedge minute(t) = 0 \}$$

⁷The set contains points in time for every day at 0:00, 6:00, 12:00, and 18:00.

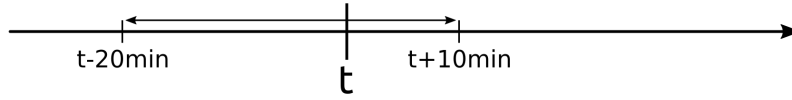


Figure 3.7: Timeline showing an allowed deviation around t of 20 minutes in the past and 10 minutes in the future. This leads to a time frame of 30 minutes where the task can be executed.

When there is an allowed deviation of 15 minutes in both directions, the set can be rewritten as follows:

$$S = \{ T \mid t \in D \wedge \text{hour}(t) \bmod 6 = 0 \wedge \text{minute}(t) = 0 \\ \wedge T = \{ t_d \mid t_d \in D \wedge t_d \geq t - 15 \text{ min} \wedge t_d \leq t + 15 \text{ min} \} \}$$

Instead of using t directly, another time variable t_d , that is arbitrarily chosen between $t - 15 \text{ min}$ and $t + 15 \text{ min}$, is (widely) defined. All t_d 's for a fixed t are contained in a set T . When e.g. a scheduling algorithm actually decides when the task corresponding to t has to be executed, the algorithm chooses one element t_d from the set T for the corresponding recurring point in time t . Using this notation, even different allowed deviations for future and past can be achieved (Figure 3.7). This is useful, for example when a task can be executed way earlier than planned but not significantly later.

Time Frames Instead of specifying one set of conditions for the elements of a periodic set, two sets of conditions are defined. These two sets create time frames in which the actual point in time for the execution of the task is located. Therefore one set specifies the beginning of the time frame and the other one the end:

$$S = \{ T \mid t_0 \in D \wedge \text{hour}(t_0) = 11 \wedge \text{minute}(t_0) = 45 \\ \wedge t_1 \in D \wedge \text{hour}(t_1) = 12 \wedge \text{minute}(t_1) = 15 \\ \wedge \text{day}(t_0) = \text{day}(t_1) \wedge \text{month}(t_0) = \text{month}(t_1) \wedge \text{year}(t_0) = \text{year}(t_1) \\ \wedge T = \{ t \mid t \geq t_0 \wedge t \leq t_1 \} \}$$

In this example t_0 defines the beginning of the time frame and t_1 the end (Figure 3.8). The periodic set describes something to happen every day between 11:45 and 12:15. As in the previous paragraph, a scheduling algorithm or something similar has to choose the actual time of execution from the set T . Please note the additional conditions $\text{day}(t_0) = \text{day}(t_1) \wedge \text{month}(t_0) = \text{month}(t_1) \wedge \text{year}(t_0) = \text{year}(t_1)$. Using this approach, it is important to specify the larger units to be equal (or in some cases successor and predecessor) for ensuring that begin and end of a time frame are not offset by multiples of these larger entities. Another way of expressing vague periodic events using time frames is specifying t_0 by conditions and specifying t_1 relative to t_0 or vice versa:

$$\begin{aligned}
S = \{ T \mid & t_0, t_1 \in D \wedge \text{hour}(t_0) = 11 \wedge \text{minute}(t_0) = 45 \\
& \wedge t_1 = t_0 + 30 \text{ min} \\
& \wedge T = \{ t \mid t \geq t_0 \wedge t \leq t_1 \} \}
\end{aligned}$$

For the examples we assumed that the smallest unit of time of the domain D is minutes. Therefore the smallest specified unit for t_0 and t_1 is minutes.

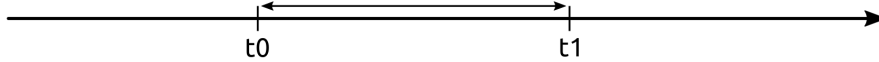


Figure 3.8: Two points in time t_0 and t_1 are creating a frame on the timeline in which the actual point in time t is located.

3.5 Implementation

Interface We decided to use a simplified version of point-vector periodicity. Our implementation makes use of the formula given in section 3.4:

$$t_i = t_0 + i \cdot \Delta t$$

To limit the degrees of freedom, we made some restrictions on Δt , t_0 and t_i .

Δt The specification of Δt is limited to the keywords “daily”, “weekly”, and “monthly”. Whereas “daily” defines Δt to be one day, “weekly” defines Δt to be seven days, and “monthly” sets Δt to one month, regardless of how many days the corresponding month has. The value of one month for Δt breaks with the assumption that the intervals of point-vector periodicity are always equi-sized. This is caused by the nature of our means of measuring time.

Limiting the domain of Δt to this set of keywords is unproblematic, because most external dependencies do not change more often than once a day.

t_0 For reasons of simplicity, t_0 is not specified by the user but set to the current date and time instead. This is also a way of achieving load balancing. Due to the fact that it is nontrivial for the user to influence the time the event occurs, the events are circadian.

t_i The value of t_i is always rounded up to a full hour. This is because the scheduler that starts the jobs only runs once an hour.

Figure 3.9: User Interface for creating a set of periodic events

We chose this implementation because Travis CI aims for being simple as well as quickly set up. Too many options would result in a slow and complicated set up process. The current user interface (Figure 3.9) needs one to six clicks to set up a cron job. Also, by limiting Δt to one day as smallest possible interval, we avoid high loads on the Travis CI infrastructure.

Using the API, a cron job can be created by sending an HTTP POST request to the `/v3/repo/{repo.id}/branch/branch.name/cron` endpoint. The payload has to consist of two key-value pairs: The first one is called `disable_by_build` and has no effect on the periodicity in terms of interval size or starting time. The `disable_by_build` key (represented with an on/off switch in the UI) makes Travis CI skip a cron build if there was another build (e.g. triggered by pushing to the version control system) during the last interval.⁸ The second key is called `interval` and accepts one of the values `daily`, `weekly`, and `monthly`. While Δt is specified via the `interval` key in the API, t_0 is always the time when the request reaches the server. Hence there is no key-value pair for this piece of information. Information about repository and branch for which the cron job has to be created are part of the URL.

Data Model The Data Model directly mirrors the interfaces. As shown in Figure 3.10 there is one database table for cron jobs that contains, besides some meta data the following relevant attributes:

branch A reference to the branch table pointing to the entry representing the branch that has to be build by the cron job.

interval Stores the interval Δt using the keywords `daily`, `weekly`, and `monthly`. This field cannot be empty and all other values are rejected by an upstream logic (i.e. the Ruby service that issues the database queries).

created_at A timestamp that represents the time when the cron job was created. This piece of information is used as t_0 in our model of periodicity.

disable_by_build A boolean value causing conditional skipping of a cron build if set to true.

⁸If the boolean value `disable_by_build` is set to true, the build triggered by the corresponding cron job is omitted if there was another build between the point in time the cron job has to run next and the last cron triggered build on this branch.

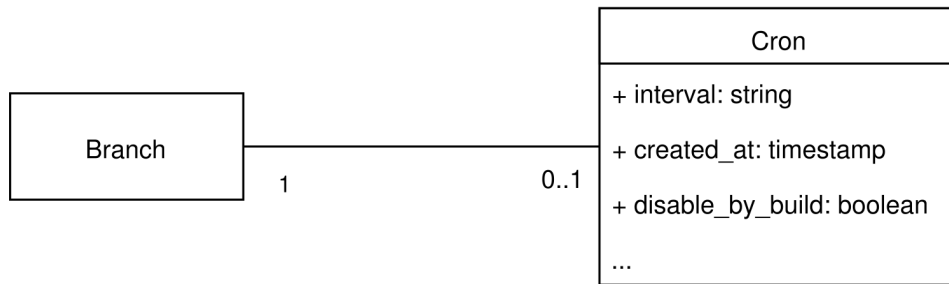


Figure 3.10: Data Model of Cron Jobs

Logic There is a service for starting all cron jobs that is triggered by a service (running on a Heroku dyno) once an hour. This service queues⁹ a cron build if the cron jobs next enqueueing time is now or already in the past (Listing 3.1).

Listing 3.1: Starting all due cron jobs

```

1 def start_all()
2   Models::Cron.all.select do |cron|
3     start(cron) if cron.next_enqueueing <= Time.now
4   end
5 end
  
```

Listing 3.2 shows how `cron.next_enqueueing` is computed. There are three points in time that have to be taken into account (Figure 3.11):

LastBuild From the current point in time going back in time to the first point that matches.

ThisBuild The current time if it matches the pattern. Else: From the current point in time going forward until the next matching point in time.

NextBuild The next matching point in time after ThisBuild.

Also, we have to distinguish three different cases. Firstly, if the `disable_by_build` flag is set, we have to check if there was another (non-cron) build after the last point in time where the cron job had to run (LastBuild, lines 2–3). In this case, this cron build (ThisBuild) is skipped and the next enqueueing is one interval later (NextBuild).

⁹The terminology uses “queueing” instead of starting, because the build is only appended to the build queue. This does not ensure that the build starts at the time of enqueueing. If there is currently high load on the infrastructure, it may take some time until the build starts.

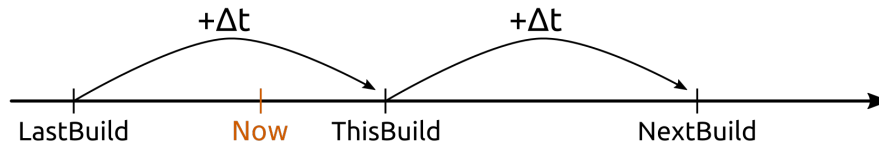


Figure 3.11: Relative position in time of build keywords

The else branch (if the condition stated above does not apply) is further divided into two cases. If there was a cron build in the last interval (between LastBuild and now), the next enqueueing is for ThisBuild (lines 4–5). If there was no cron build in the last interval, the cron job is overdue and has to be enqueued immediately (lines 6–7).

Listing 3.2: Choosing the correct element of a periodic set

```

1  def next_enqueueing
2    if disable_by_build && last_non_cron_build_date > planned_time(LastBuild)
3      planned_time(NextBuild)
4    elsif last_cron_build_date >= planned_time(LastBuild)
5      planned_time(ThisBuild)
6    else
7      Time.now
8    end
9  end

```

By now, the logic for selecting the appropriate element of the periodic set is clear. A topic yet to be discussed is, how the considered elements of the periodic sets are generated. To achieve the generation of DateTime objects out of the keywords LastBuild, ThisBuild and NextBuild, we need to distinguish between “daily”, “weekly”, and “monthly” intervals. This is because the implementation generates DateTime objects not by using the formula $t_i = t_0 + i \cdot \Delta t$ but by assembling it from its components (e.g. year, month, day, hour, ...), as already described in section 3.4.1. Therefore, the only thing `planned_time` does is selecting a method depending on the value of interval (Listing 3.3).

The methods `planned_time_daily`, `planned_time_weekly`, and `planned_time_monthly` assemble DateTime objects for ThisBuild. Afterwards the DateTime object is offset by one day, week, or month to future or past. Since the keywords LastBuild, ThisBuilds, and NextBuild are internally represented as integer constants with values -1 , 0 , and $+1$ (Listing 3.4), they can be used as factors for the offset.

For the assembly of the ThisBuild DateTime object two DateTime objects, `DateTime.now` and `created_at`, are used. For a “daily” interval the year, month, and day

Listing 3.3: Choosing the algorithm depending on the value of interval

```

1 def planned_time(in_builds = ThisBuild)
2   case interval
3   when 'daily'
4     planned_time_daily(in_builds)
5   when 'weekly'
6     planned_time_weekly(in_builds)
7   when 'monthly'
8     planned_time_monthly(in_builds)
9   end
10 end

```

Listing 3.4: Values of the keywords LastBuild, ThisBuild, NextBuild

```

1 LastBuild = -1
2 ThisBuild = 0
3 NextBuild = 1

```

attribute are taken from `DateTime.now` while the hour attribute is determined by `created_at` (line 3 in Listing 3.5). Smaller units of time (e.g. minutes and seconds) are omitted and therefore implicitly set to zero. Afterwards, there is a correction by adding one day if the cron job should have already run today. In this case `ThisBuild` is tomorrow (line 4). The last step in `planned_time_daily` is adjusting the `DateTime` object by adding or subtracting one day if `NextBuild` or `LastBuild` is requested instead of `ThisBuild` (line 5).

In the following code snippets the `+` operation between `DateTime` objects and `Integers` returns a `DateTime` object with a number of *days* specified by the `Integer` added to the original `DateTime` object. The `>>` operation between `DateTime` and `Integer` is similar but adds *months* instead of days to the original `DateTime` object [39].

Listing 3.5: Algorithm for assembling `DateTime` object with a daily interval

```

1 def planned_time_daily(in_builds)
2   now = DateTime.now
3   build_today = DateTime.new(now.year, now.month, now.day, created_at.hour)
4   return build_today + 1 + in_builds if (now > build_today)
5   build_today + in_builds
6 end

```

The algorithm for weekly intervals (as shown in Listing 3.6) basically does the same as the one for daily intervals but rounds up the day part of the `DateTime` object to the next fitting weekday (line 4). Also, the 1-day corrections for daily intervals become 7-day corrections for weekly intervals (lines 5–6).

Listing 3.6: Algorithm for assembling DateTime object with a weekly interval

```

1 def planned_time_weekly(in_builds)
2   now = DateTime.now
3   build_today = DateTime.new(now.year, now.month, now.day, created_at.hour)
4   next_time = build_today + ((created_at.wday - now.wday) % 7)
5   return build_today + 7 * (1 + in_builds) if (now > next_time)
6   next_time + 7 * in_builds
7 end

```

For monthly intervals, the DateTime object for ThisBuild is assembled by calculating the month difference between now and the creation time (lines 5–6). This month difference is added to the creation time which results in the point in time the cron job had to run this month (line 7).¹⁰ Afterwards it is checked if this point in time is in the past (the cron job already had to run this month). If so, one month is added for correctly providing ThisBuild. Finally, `in_builds` is used for correction by one month if NextBuild or LastBuild is requested instead of ThisBuild.

Listing 3.7: Algorithm for assembling DateTime object with a monthly interval

```

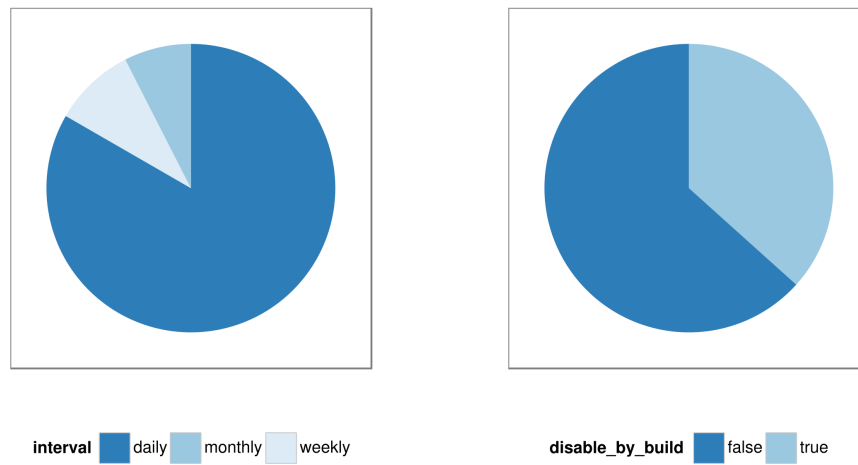
1 def planned_time_monthly(in_builds)
2   now = DateTime.now
3   created = DateTime.new(created_at.year, created_at.month, created_at.day,
4                          created_at.hour)
5   month_since_creation = (now.year * 12 + now.month)
6                          - (created_at.year * 12 + created_at.month)
7   this_month = created >> month_since_creation
8   return created >> (month_since_creation + 1 + in_builds) if (now > this_month)
9   created >> (month_since_creation + in_builds)
10 end

```

3.6 Evaluation

With our implementation of periodic task schedules we tried to achieve two main goals: Simplicity for the users as well as load balancing to avoid bursts on the Travis CI infrastructure.

¹⁰If the cron job has to run on a day that is not existent in the current month (for example there is no 31st June), the last day of the month (e.g. 30th June) is selected. This strategy is part of the implementation of the Ruby Date class.



(a) Ratio of intervals chosen by users (b) Share of cron jobs with `disable_by_build` flag

Figure 3.12: User choices concerning cron jobs

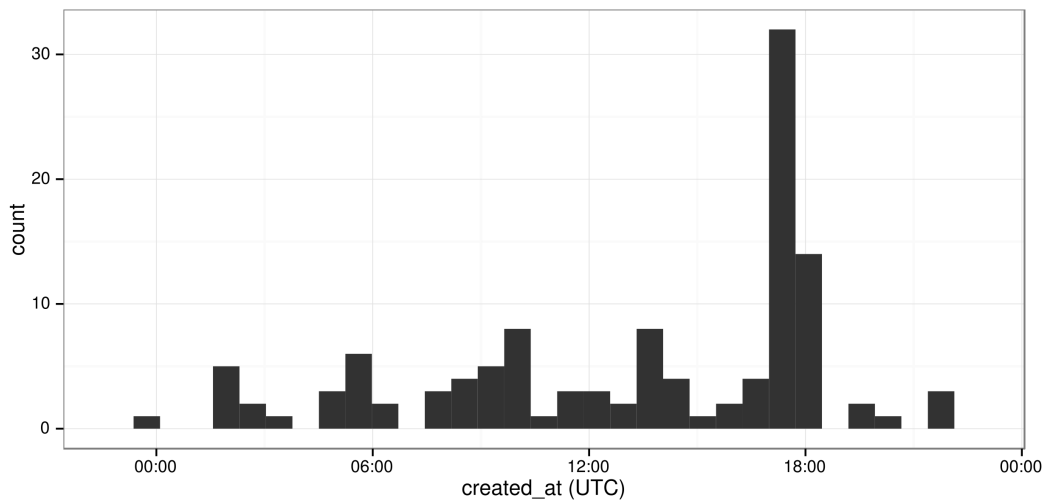


Figure 3.13: Histogram of `created_at` times

Simplicity We tried to achieve simplicity by providing only a minimal set of configuration options. The user has to choose only the branch that has to be addressed by the periodic task, the interval size and if a cron job should be skipped if there was another build (e.g. triggered by a commit and push) during the last interval (`disable_by_build` key). After making the cron feature open to the public, we figured out some users do not understand the meaning of the `disable_by_build` key. We had issues describing the effect of enabling/disabling this key in an understandable manner while keeping the explanation text short so it does not break the user interface. However, as Figure 3.12b shows, most users disable the key although it was enabled by default. This draws the conclusion that either most users understood the meaning since they actively decided against using this feature or the users just disabled the key because they did not know what it does and therefore did not want it to be enabled.

Another issue in the user interface is the way of displaying when the cron build is enqueued. As long as a cron job is not due, the due time is displayed. In contrast, when a job is overdue, the current time is displayed to express that the job has to be scheduled right now. While this approach is suitable for the scheduler, it is confusing for users since they expect the job to start now. Unfortunately the service starting the jobs runs only once an hour. Therefore jobs can be overdue for up to one hour which leads to counting due times over a relatively long period of time which is confusing for users.

Load Balancing To avoid bursts on the Travis CI infrastructure we tried both reducing and balancing the load produced by periodic tasks. Load reduction was tried to achieve by setting the default configuration of the user interface so that it produces the least load. Therefore “monthly” is the default option for the interval size and the `disable_by_build` key is set to true. As the charts in Figure 3.12 show, the users are showing a greedy behavior. Chart 3.12a expresses that “daily” is the most common interval option while “weekly” and “monthly” are rarely used. There are even users requesting hourly running cron jobs. This option is however not provided due to capacity issues. Congruously, chart 3.12b shows the `disable_by_build` key to be disabled in approximately two thirds of the cron jobs. Another mean of load reduction is limiting the number of cron jobs to one cron job per branch. This cannot be influenced by the user and is enforced on database level.

In addition to the attempt of load reduction, load balancing is applied to cron jobs. The mean of distributing cron jobs equally over the day is not letting the user specify the time a cron job has to run but using the creation time. This implies that a cron job runs daily, weekly, or monthly at its time of creation. Since the users of Travis are distributed around the globe and hence in different time zones which leads to different work times, the creation times are distributed over the day and thus the enqueueing times are also distributed. Figure 3.13 shows that this concept works well, except for the peak around 18:00.

At the time of writing this chapter, there were a total of 120 cron jobs. With respect to that it is too early to draw meaningful conclusions out of this data. Currently, the chosen approach seems to be sufficient while not perfect.

There is another advantage of using the creation time as the enqueueing time (called t_0 in the mathematically motivated point-vector model), namely that the user is able to influence the creation time while not being encouraged to do so. This implies that if a user has an important need to run a cron build at a specific time the user can just create the cron job at this time (of the day, week, or month). If the user does not care about the time of enqueueing (which we expect to be the default case), he can just create the cron job without considering the time.

3.7 Related Work

The idea of periodic schedules was already topic of other works. The two paradigms of expressing periodic schedules presented in this work are actually used. They are combined (cf. section 3.4.3) or modified when in use, though.

3.7.1 Ruby/RUNT

Martin Fowler proposed an approach for implementing pattern based recurring events [20]. While the way of dealing with recurring events described in Fowlers work shows strong similarities with the conditional periodicity proposed in this chapter, the work by Fowler has a stronger focus on the implementation. This approach was later implemented as the RUNT (Ruby Temporal Expressions) package for Ruby.¹¹ The concept of RUNT is to create expressions that can be matched by a Date or a DateTime object. Listing 3.8 shows how temporal expressions can be created. More complex expression can be created by the use of conjunctions (&) and disjunctions (|). In Listing 3.9 an expression matched by each Tuesday afternoon is constructed out of an expression matched by Tuesdays and an expression matched by afternoons. Also, an expression matched by Tuesdays and Thursdays is created. To check, if a Date or DateTime object matches the expression, the method `include?` is defined for RUNT expression objects. Listing 3.10 depicts the usage of `include?` [31, 32].

Listing 3.8: Creating temporal expressions with RUNT in Ruby

```
1 tuesdays = DIWeek.new(Tue)           # every Tuesday matches
2 afternoons = REDay.new(12,0,18,0)    # between 12:00 and 18:00
3 octobers   = REYear.new(10)          # 10th month of the year
```

¹¹<https://github.com/mlipper/runt> (visited on 2016-06-21).

Listing 3.9: Combining temporal expressions using conjunction and disjunction

```

1 tuesdays    = DIWeek.new(Tue)
2 thursdays  = DIWeek.new(Thu)
3 afternoons  = REDay.new(12,0,18,0)
4
5 tuesday_afternoons = tuesdays & afternoons
6 tuesdays_and_thursdays = tuesdays | thursdays

```

Listing 3.10: Checking if a Date (or DateTime) object matches the expression

```

1 octobers      = REYear.new(10)
2 day_in_march = Date.new(2016,3,10)
3 day_in_october = Date.new(2016,10,3)
4
5 octobers.include?(day_in_march)    # => false
6 octobers.include?(day_in_october)  # => true

```

3.7.2 Microsoft

In 1996 the Microsoft Corporation together with Redmond and Wash requested a patent [41] for another approach of representing recurring events. Their approach is designed for high expressiveness and was originally built for calendar software. There are strong similarities between the solution by Microsoft and the hybrid periodicity type introduced in this work (section 3.4.3). The periodicity format by Microsoft features equi-sized intervals in the manner of point-vector periodicity combined with time range limitation and other conditionals as depicted in section 3.4.4. The primary issue with the format presented in the patent is the high complexity which makes it unsuitable for end-user interfaces while being still convenient as interface for experts or as API. The patent mainly focuses on the data structure required by this approach, but also presents an algorithm for evaluating recurring events using this approach. The data structure introduced by the patent contains 15 fields:

start date Specifies the start of the period the recurring event is defined for.

end date Specifies the end of the period the recurring event is defined for.

recurrence type A bit mask stating which of the following seven fields are to be evaluated. Each of the following field is evaluated if and only if the corresponding bit in the recurrence type field is set to 1.

day interval If enabled, indicates the number of days between the occurrences of the recurring event. The first occurrence is at the *start date* defined above.

week interval If enabled while *month interval* and *month of year* are disabled, indicates the number of weeks between the occurrences of the recurring event. The

3 Interfaces for Expressing Periodic Task Schedules

first week is then by definition the week the *start date* is in. If enabled while either *month interval* or *month of year* is also enabled, indicates which week of the month is affected by the recurring event. When using any month key with the *week interval* key, there is only one week per month where occurrences can be in. There are no occurrences in another week of the month.

month interval If enabled, indicates the number of month between the occurrences of the recurring event. First month is the month where *start date* is in.

year interval If enabled, indicates the number of years between the occurrences of the recurring event. First year is the year where *start date* is in.

day of week A bit mask of seven bits stating which days of the week occurrences are at. Needs to be enabled by the *recurrence type* field.

day of month A bit mask of 31 bits stating which days of the month occurrences are at. Needs to be enabled by the *recurrence type* field.

month of year A bit mask of 12 bits stating which months of the year occurrences are at. Needs to be enabled by the *recurrence type* field.

start of week A day of the week the week starts with.

start time Specifies the start of the time frame at a given day for each occurrence of the recurring event.

end time Specifies the end of the time frame at a given day for each occurrence of the recurring event.

timezone States the time zone times are expressed in.

description A human readable description of what is expressed with the fields described above.

Note that there are combinations of fields in use that are not meaningful. For example a combination between *month interval* and *month of year* leads to overspecification of the month part which raises the question of whether the first or the latter specification is more important. The behavior for this case is undefined. Therefore only some bit combinations are permitted for the *recurrence type*.

The algorithm evaluating if a given date contains an occurrence of the recurring event works top down. The checks performed are (in this order):

1. Does the year match the record?
2. Does the month match the record?
3. Does the week match the record?
4. Does the day match the record?
5. Are there any exceptions (like overspecification problems described above)?

The whole algorithm uses days as smallest unit of time to iterate over. The time part is generated after the checks using the fields *start time* and *end time*. This is performed in GMT with a time correction transforming GMT to the time zone stated in the *time zone* field afterwards.

3.7.3 Android

The Android operation system provides two means of dealing with periodic tasks [2].

Alarm Type There are two different approaches of dealing with periodic tasks. These approaches are called “alarm types” in context of Android and are implementations of the point-vector periodicity presented in this section 3.4.1. The first alarm type is called “elapsed real time”. Android is counting the time since it booted. This time is used as continuous and linear reference time that abstracts from issues like time zone changes, calendrical inconsistencies¹² and leap seconds. A developer using this implementation of periodic schedules only has to define an interval (in milliseconds) and a task (referenced by an Intent) that has to be executed. In the point-vector model t_0 is then the “elapsed real time” (the time since boot) at the time when the periodic task schedule is issued. A fixed time-offset can be applied to t_0 , though (Listing 3.11). The value of Δt is the interval defined by the developer and i is a in \mathbb{N}_0 and can therefore not be negative since every point in time before t_0 already passed. When the system reboots, the periodic task schedule has to be reissued which leads to a reset of t_0 . Thus values for t_i smaller than t_0 can not be reached.

The second alarm type is “real time clock” which means UTC wall clock time. This is used when execution at a specific time of a specific day has to be achieved. In contrast to “elapsed real time” not only the interval size Δt can be specified, but also the starting point t_0 (Listing 3.12).

Listing 3.11: Creating exact recurring events starting in 10 minutes, repeating every half hour. `AlarmManager.INTERVAL_HALF_HOUR` is a constant containing the value 1800000 which is half an hour expressed in milliseconds.

```
1 alarmManager.setRepeating(AlarmManager.ELAPSED_REAL_TIME,
2 10 * 60 * 1000, // starting in 10 minutes (offset to current time)
3 AlarmManager.INTERVAL_HALF_HOUR, // repeating every half hour
4 someIntent);
```

¹²The major inconsistencies in calendars are connected with the number of days in a month. There is a need to know the month to conclude how many days the month contains. Another inconsistency is that weeks and months are both units of time in use without being connected to each other. Therefore there is no way to conclude from the day of the month to the day of the week or vice versa.

Listing 3.12: Creating exact recurring events using “real time clock” alarm type. A starting point t_0 is manually specified by using the current time and manipulating the hour part.

```
1 // specify date and time for today at 17 o'clock
2 Calendar calendar = Calendar.getInstance();
3 calendar.setTimeInMillis(System.currentTimeMillis());
4 calendar.set(Calendar.HOUR_OF_DAY, 17);
5
6 // create recurring event that fires today at 17 o'clock and repeats every day
7 alarmManager.setRepeating(AlarmManager.RTC,
8     calendar.getTimeInMillis(),
9     AlarmManager.INTERVAL_DAY,
10    someIntent);
```

Wake Up Next to choosing the alarm type, the developer has to specify if the recurring event has to fire if the device sleeps. Firing if the device sleeps is called “waking up” the device. If “waking up” is enabled, the alarm will fire at the time it is scheduled even if the device is sleeping. To enable “waking up”, the keyword `ELAPSED_REAL_TIME` has to be replaced with `ELAPSED_REAL_TIME_WAKEUP` and the keyword `RTC` has to be replaced with `RTC_WAKEUP`, respectively. With disabled “wake up” option and a device currently sleeping, the alarm is postponed until the next awaking.

Precision The scheduler for recurring events in Android offers two different modes for dealing with precision. Firstly, there is the `setRepeating()` method that issues a recurring event with exact scheduling without leaving room for optimization. Secondly, the `setInexactRepeating()` method can be used for creating recurring events that can be optimized by the scheduler. The events are then scheduled earlier or later with the goal to run many recurring events together. This reduces drain on the battery due to fewer wake ups. When using `setInexactRepeating()`, only some values for the interval are permitted. A list of permitted interval sizes can be found in the reference manual of the Android `AlarmManager` [3]. It is nonetheless recommended to use `setInexactRepeating()` to avoid battery drain.

3.8 Conclusion

There are different ways of expressing periodic task schedules. When trying to choose an appropriate solution for a specific use case, different approaches have to be taken into account. In this work we presented two paradigms of expressing periodic schedules, point-vector periodicity and conditional periodicity, while describing advantages and disadvantages in terms of understandability, precision, and expressiveness. With focus on understandability we decided to use a simplified ver-

sion of point-vector periodicity in our solution. The implementation and issues we ran into are also subjects to this work. There are formats for recurring events that are different from the solutions we presented. Nonetheless these approaches are often attributable to the presented paradigms or are hybrid form of them.

4 Dependency Management for Hosted Continuous Integration Services

Dependencies are an integral part of modern software development. Unstable dependencies can potentially alter or even break applications at any time. Testing software whenever its dependencies are changed ensures that the developers immediately learn about any such impact so they can make the necessary adjustments. This chapter proposes a feature addition to the continuous integration service Travis CI which accomplishes that. It lets users create dependency relations between projects tested on Travis CI and build up dependency graphs. The finishing of a build triggers the dependants to be tested, forming build sequences along these graphs. The resulting information is presented using status badges and helps developers identify external problem sources faster.

4.1 Introduction

Dependencies have become almost inevitable in software development. Unstable dependencies pose a constant risk because they can transform over time. Such changes are capable of having drastic effects on their dependants [16, 43]. Knowing about any such repercussions is time critical.

This problem has in part been solved by a number of continuous integration (CI) tools through *build pipelines*, but only for dependencies between modules of the same project or organization [23, 24, 28]. We aim to bring this approach to the hosted CI service Travis CI. This would open up the possibility to declare dependency relations with external software tested on Travis.

GitHub is a social network for programmers, hosting over 38 million project repositories [22], of which many are interrelated and through which users can connect and reuse each other's code. Travis is tailored to integrate seamlessly into GitHub and allows users to automatically run their software tests whenever they push changes or make a pull request.

We propose an additional feature for Travis that notifies users whenever their application has been affected by changes made to its dependencies.

Our solution enables developers to create dependency relations between GitHub repositories and build up dependency graphs. This way, whenever a dependency has been updated, its dependants are automatically tested on Travis, resulting in what we call build sequences. The results of these tests are presented through status badges on the repositories' GitHub page for the status of dependencies and dependants, as well as visualisations on the Travis website, helping programmers keep control over their dependencies.

4.2 Background

According to the Oxford Dictionary, a dependence is “*the state of relying on [...] someone or something else*” [36]. The one that is relied upon is called the *dependency*, the other is called the *dependant*.

This concept plays a major role in software architecture today, because when working on modern software projects, most developers rely on already existing technologies. They use premanufactured hardware, program on top of existing software like operating systems, rely on common practices, and reuse already written source code that can either stem from themselves or from external sources. This radically reduces the efforts needed to create the desired piece of software, but creates the aforementioned state of reliance, which can sometimes have many far reaching complications.

Most software projects incorporate more than just one dependency, and dependencies often have multiple dependants. The dependency graph in Figure 4.1, in which each node represents a direct or indirect dependant of the npm package *async*¹ illustrates this. This means that the changes made to a depended upon project affects many others, in the case of *async* at least 25 % of all npm packages [43]. These numbers are from July 17, 2013, and since *async* is still in active development with currently (July 5, 2016) over 18,300 *Stars* on GitHub, they have probably already increased a lot. A recent example showing how large dependency graphs can be problematic were the disastrous consequences of the unpublishing² of the npm package *left-pad*'s repository from GitHub by its owner Azer Koçulu (*azer* on GitHub) because of a copyright dispute. *Left-pad* consists of only eleven lines of code, yet it was being used by thousands of other projects, which all broke instantly. Affected were also large scale projects like *React*³ which is developed by Facebook and Instagram and used e.g. by billion dollar companies such as Netflix and Airbnb.

Explicit and Implicit Dependencies

Dependencies can be divided into two different types. *Explicit* dependencies are external code (e.g. software libraries), that is incorporated directly into a project. Before being able to reuse such code, the developers have to explicitly declare the dependency. This is done either directly within the code (e.g. by adding `#include <libraryname>` to a C file), or in dedicated dependency specification files (e.g. the *Gemfile*⁴ in Ruby, see Listing 4.1).

Implicit dependencies are usually autonomous entities that the dependant interacts with. They can also be referred to as *infrastructure dependencies* [43] and take on many different forms. They are not explicitly specified within a project's code, but removing them from the setup or changing them could result in erroneous behavior or even make it entirely impossible to use the dependant. Examples would be a

¹<https://github.com/caolan/async> (visited on 2016-07-05).

²<https://medium.com/@azerbike/i-ve-just-liberated-my-modules-9045c06be67c#.nttc9x78y> (visited on 2016-07-05).

³<https://facebook.github.io/react/index.html> (visited on 2016-07-05).

⁴<http://bundler.io/v1.3/man/gemfile.5.html> (visited on 2016-05-25).

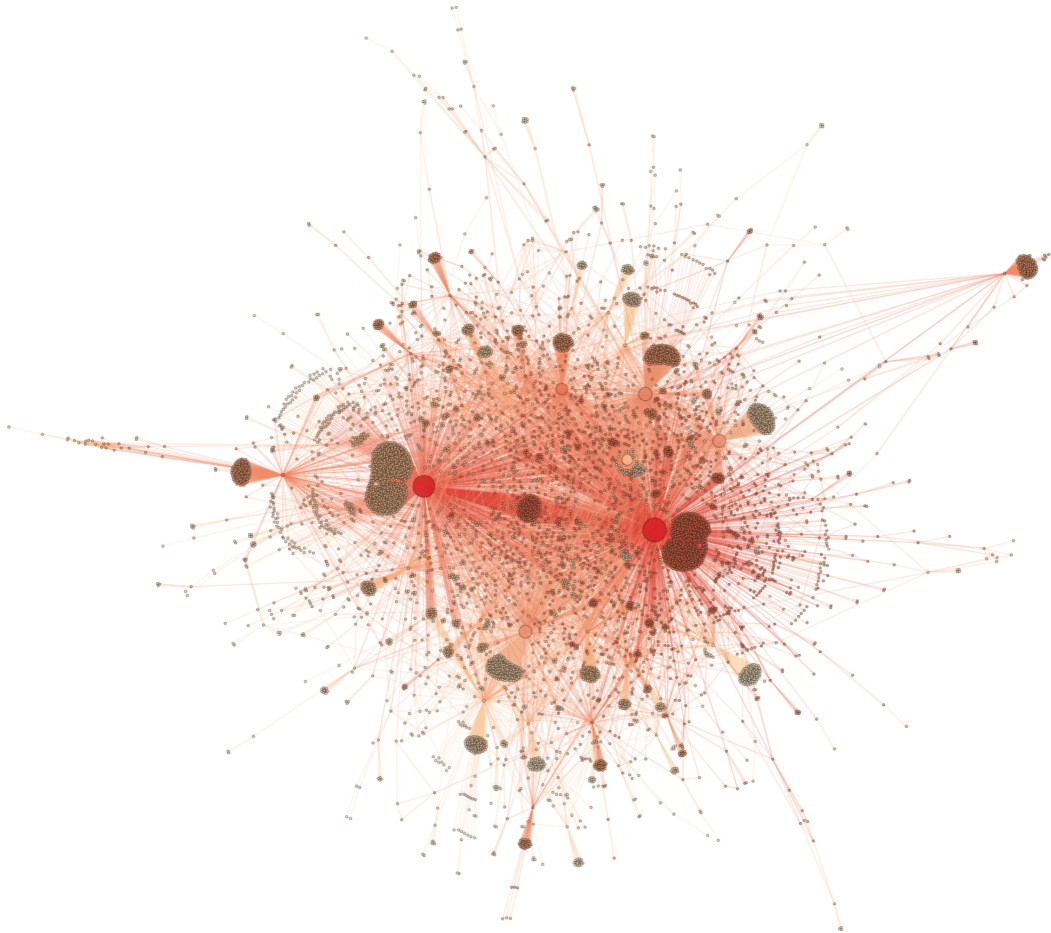


Figure 4.1: "The dependency graph of the npm package *async* (red, middle right). The graph has a total of 8 797 nodes and 13 725 edges" [43]

need for an operating system that supports the POSIX standard or a web API that the dependant needs to contact constantly, which therefore has to always be online. Implicit dependencies can also be a certain type hardware, but in this chapter we will only focus on software dependencies.

Any dependency, be it explicit or implicit, can for different reasons be called *unstable*. This is the case if it is possible for the dependency to change in some way at any time.

Explicit dependencies are unstable if they are *loosely specified*. The following example illustrates this. A Gemfile like the one in Listing 4.1 can describe a Ruby project's explicit dependencies in two different ways: They are *tightly specified*, if they have a fixed version next to them (see Listing 4.1, Lines 3 & 4), and they are *loosely specified*, if they do not. Instead it could be declared that the project supports anything newer than a certain version (see Listing 4.1, Line 5), or by only naming their GitHub repository (see Listing 4.1, Lines 6 & 7), which always refers to the dependency's latest available version. Since they can over time point towards different states of the dependency, such loosely specified dependency declarations are unstable.

An unstable implicit dependency like an online service could go offline, or another module in a modular architecture could for example suddenly change its behavior.

Listing 4.1: An example Gemfile for a fictional Ruby project

```
1 source 'https://rubygems.org'
2
3 gem 'activesupport', '4.0'
4 gem 'metriks',      '0.9.9.6'
5 gem 'nokogiri',    '>= 1.4.2'
6 gem 'travis-core', github: 'travis-ci/travis-core'
7 gem 'yard-sinatra', github: 'rkh/yard-sinatra'
```

For developers, unstable dependencies can sometimes be very desirable, as the changes made to them are usually meant to be enhancements, like improved speed or memory usage. Thus having the reused technologies on the *bleeding edge*⁵ can be advantageous but, because of their unpredictability, also cause problems. These are in general not knowing when or whether the dependency will or has been updated, not knowing what has been changed, and having to adjust one's software whenever the changes have had unwanted impact.

In the following pages we will discuss our solution of this problem which combines dependent builds with the structural advantages of hosted continuous integration systems that are integrated into repository hosting services, focusing on Travis CI in particular. However, the concept and its benefits could be transferred to any other hosted CI service.

⁵"The most advanced stage of a technology [...], usually experimental and risky" [14]. Using the bleeding edge of a technology thus refers to always employing the latest version.

4.3 Motivation

A common practice in continuous integration is for developers to test their entire software system after each integration, which is a key feature of hosted CI services like Travis. But testing on own code changes alone is insufficient, since, as we have examined above, changes to the dependencies can also alter the dependant systems' behavior. Therefore running tests after changes made to their dependencies would prove useful for many software projects.⁶

To a degree, continuous integration tools like Jenkins and TeamCity have already solved this problem by letting users define so called *build pipelines*. However, since these tools only operate inside the context of a single organization or project, the set of selectable dependencies is constrained. It follows that dependency graphs defined using these tools cannot include external software, which makes them unable to solve the problems presented above, since these are caused by external dependencies, which the developer does not have any control over.

It would furthermore be of interest for developers who use hosted CI systems to integrate dependency support within the global infrastructure of that service, notably the synergic complex of GitHub and Travis CI, and their large user base. The 15 million GitHub users with their 38 million repositories and the more than 300,000⁷ of these that use Travis CI [22], could benefit from creating dependency graphs of them.

Throughout this chapter we will use Travis as an example project, on which we will base the needed requirements for a useful dependency management feature. Figure 4.2 shows a small fraction of the software dependencies within the architecture, with some fictional modules and dependency relations added.⁸

Travis' infrastructure is based on several software modules that are all interconnected and hence depend on each other. They all have a GitHub repository and since most of them are also tested using Travis CI itself, it would be convenient for them to have a test pipeline according to their dependency graph. The developers at Travis CI would be notified if a change to one of their modules has broken another one and they could immediately fix the problem, instead of being informed about it by a bug report from a user after an indefinite amount of time.

Some of these modules also depend on external software such as Sinatra⁹ which is hosted in a GitHub repository itself. Say one module has Sinatra in their Gemfile, pointing to the GitHub repository, which is a loose specification, since Bundler will always clone and then install the gem's current state of the project's master branch (if

⁶A different solution to this problem, namely the use of periodical testing, is discussed in chapter 3.

⁷Taken from the repository counter on <https://travis-ci.org> (visited on 2016-06-10).

⁸The fictitious repositories `travis-lib` and `travis-web-mobile` were included because the design decisions that would lead to certain problems addressed within this section were not present in Travis' actual architecture.

⁹<https://github.com/sinatra/sinatra> (visited on 2016-06-14).

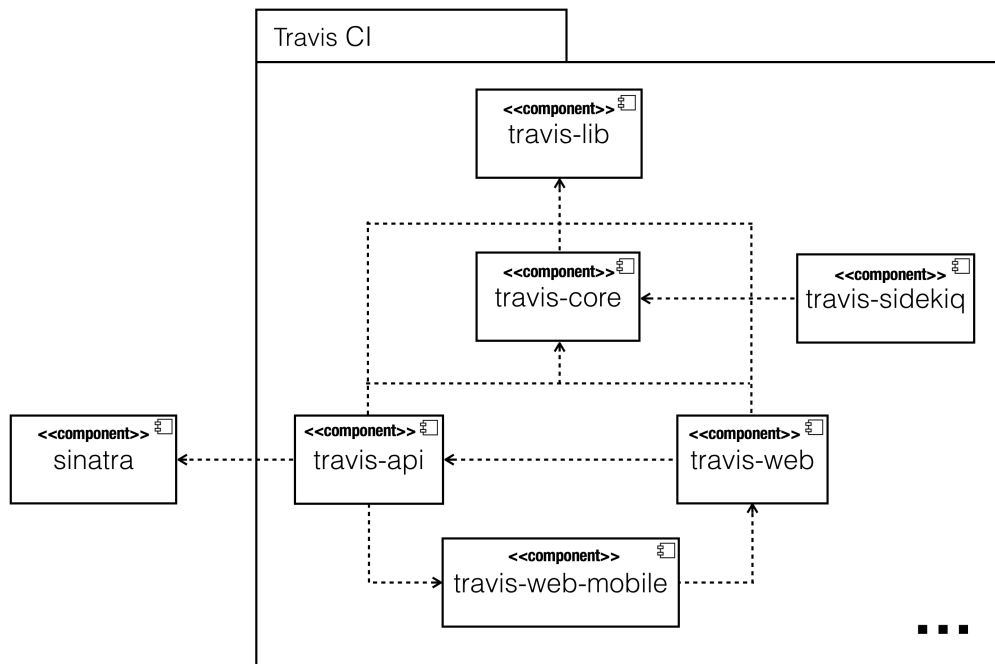


Figure 4.2: An excerpt of the complex dependency system in the Travis CI infrastructure with some added details, represented through UML syntax.


no other branch is specified)¹⁰ If someone wanted to clone or fork that module, and they see that its latest Travis build was successful, they should be able to assume that a local test on their own computer would also be successful. But even though the current state of the codebase equals that from the latest test (which could have been done long ago, because nobody has been working on it), the local test could still fail because of that loosely specified dependency. Since that last build on Travis, Sinatra could have in the meantime been changed substantially. Its latest version could break the module and Travis CI would be oblivious to the incompatibility. This can cause all sorts of problems and frustration for both the developers of Travis and the people of the open source community. It would therefore be very helpful for developers to know that a change to one of their dependencies has broken their project.

The Missing Feature

A feature request for dependent builds has been made via the GitHub Issue tool (see Figure 4.3), which shows that the community does have a certain interest in such a feature. It is now closed because the idea has been put on the list of features that Travis CI wants to add to its service. Moreover, the various workarounds that users have come up with over the time to create dependent builds demonstrates the usefulness of such a feature.

¹⁰<http://bundler.io/git.html> (visited on 2016-06-14).

Dependent builds #249

 **Closed** dkubb opened this issue on Sep 7, 2011 · 23 comments



dkubb commented on Sep 7, 2011

An awesome feature to have would be a way to specify project dependencies (or better yet infer them from the Gemfile/gemspec), so that when a project builds successfully, all it's dependents' (that travis knows about) are built too.

This is a feature we use in the DataMapper CI, where anytime dm-core passes, it executes a build for every other dependent gem.

Figure 4.3: The feature request on GitHub for dependent builds. The conversation that followed can be read in full at <https://github.com/travis-ci/travis-ci/issues/249> (visited on 2016-05-30).

A noteworthy example was implemented by the company RightScale.¹¹ Their solution identifies a build as belonging to a repository that is a dependency to others and then runs a custom shell script called `trigger-dependent-build` to test the dependants. To get notified about a resulting failed build, they employ Slack notifications.¹² Others made use of the possibility to include shell commands in the `.travis.yml`, through which they triggered builds for their dependant repositories.¹³ Nevertheless, an implementation of dependency support that is directly integrated into the Travis CI infrastructure has not been attempted yet.

For these reasons we have designed a solution to add a dependent builds feature to Travis.

4.4 Approach

Throughout this Section, we will present concepts behind our proposed dependent builds feature for Travis CI. We start out with a very general description of its functionality, followed by an abstract view on how to test the compatibility of interdependent modules in dependency graphs. The rest will be detailed considerations of all aspects of the design, describing how it works and how the users would interact with it. A key component of the design is a new module for the Travis CI architecture named *DependencyManager*, whose behavior is described in depth in the passage *Running a Build Sequence* of this Section.

¹¹<https://eng.rightscale.com/2015/04/27/dependent-builds-in-travis.html> (visited on 2016-05-30).

¹²<https://api.slack.com/incoming-webhooks> (visited on 2016-05-30).

¹³An example for this can be found at <https://github.com/mernst/plume-lib/blob/master/bin/trigger-travis.sh> (visited on 2016-05-30).

4.4.1 Basic Requirements

At large, the proposed feature addition for Travis CI should enable the users to establish dependencies between repositories to test for errors resulting from changes to the dependencies of a project. It should be completely integrated into the already existing modular infrastructure of Travis CI.¹⁴ A user should be able to conveniently create a directed relation \succ between two GitHub repositories A and B that both use Travis CI. This relation $A \succ B$ then represents a dependency of repository A from B (see Figure 4.4) which can, depending on the context, either be explicit or implicit. If such a relation has been declared, then it should follow that whenever B is tested in Travis CI, its dependant A will be tested subsequently using the new version of the dependency.

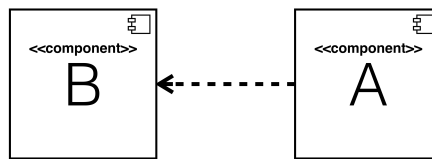


Figure 4.4: A simple dependency relationship of a software module A from a software module B in UML syntax

What follows from these dependency declarations is the feature’s capability to let the user create dependency graphs based on their project’s dependency infrastructure, like the one shown in Figure 4.2. The successive testing of the repositories in the dependency graph’s subgraph with the first built project at the root will be called *build sequence* within this chapter.

For developers, the most important information that results from this will be that a build fails because of a change to a dependency. If this happens, they should therefore be notified immediately. It would also be helpful to visualise the build sequence on the Travis website to show the build chain that made a dependant’s test fail. Then the developer is able to easily identify the code changes that caused the errors, as the build chain can be traced back to a definite push to GitHub.

Another important and widely used feature of Travis is the testing of unmerged pull requests. Testing dependants for compatibility with the dependency in the state after the pull request merge is therefore also required.

4.4.2 Testing of Dependencies

On the most basic level, testing a software with an architecture based on interdependent modules whenever one of them has changed can be accomplished by trans-

¹⁴See chapter 2 for a detailed description of the architecture of Travis CI.

forming the dependency graph to a tree¹⁵ starting from the just updated module and then kicking off a *build sequence*. This means that whenever a module has been built and tested, its dependants will then also be tested, but always using the new resulting version of all dependencies in the graph.

Since dependencies can be both explicit or implicit, they vary greatly in their nature. It is hence important to consider what dependency compatibility testing has to look like, so that changes pushed to a dependency could potentially cause a dependant's tests to fail. In both cases the most important prerequisite is of course that the dependencies are somehow usable by the dependant's tests. To achieve this, Travis Build will need an installation script for each node from the build sequence tree above the dependant, which it concatenates in the build order and then injects into the build script.¹⁶ These scripts have to differ depending on the dependency type, which the user therefore always has to specify.

Explicitly declared dependencies (e.g. software libraries) are only utilised for code reuse, which means that their code is employed directly within the dependant project's code. But for certain programming languages just cloning a dependency's repository does not suffice to make it usable. In the case of C++ for example, it will have to be compiled and stored as a dynamically linked library. Hence the installation scripts for explicit dependencies will need to be language specific. They should usually equal or be very close to the language's normal build script without the actual testing and be provided by Travis.

Implicit dependencies (e.g. an API that is deployed on a server) on the other hand take on many different roles within software architectures. They can either take the form of a program running next to the dependant on the same machine, or they can be deployed on an external server. Testing the support of a dependency of the first type can again look very different depending on the individual context. The needed setup script can therefore not be supplied by Travis, but has to be provided by the user by having it in the dependant's GitHub repository.¹⁷

This way any sort of integration work that could be needed would be accommodated for. Such scripts would for example include the setup of a local surrogate for the implicit dependency and changing of certain variables to contact it instead of the web resource. They could in theory be used for any possible setup and thus grant much power to the users, but also transfer responsibility to them to employ them with care.¹⁸ It could albeit become problematic if the user provided scripts are not working, or doing nothing but waiting for an event that may or may not occur, but

¹⁵In our solution this is done in part when storing the dependencies in the database and in part during execution of the build sequence (see Section 4.4.3).

¹⁶When installing something, these scripts should first check whether it has already been done before. In such cases the item has been installed earlier during the execution of another dependency's script and should thus not be overwritten but used instead.

¹⁷To put these scripts in the dependant's `.travis.yml` would not be suitable, since this would clutter the file with scripts for different dependencies and contexts.

¹⁸Making a deploy to Heroku part of the script could for example be problematic, since in cases like pull request testing, the script would deploy an unmerged branch. Such considerations should be mentioned in the feature's documentation.

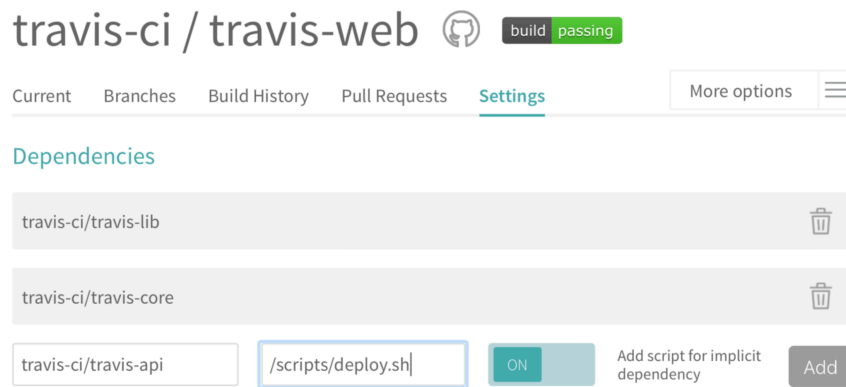


Figure 4.5: Design mockup for dependency creation in the settings page on the `travis-ci.org` website, with entries being made corresponding to `travis-web`'s dependencies

Travis CI is already adapted to such cases and stops the job when for example the log has not produced any output for a certain time.

After a Travis Worker has executed the (concatenated) setup script for the testing environment to include all the dependencies' latest versions the actual testing can begin.

4.4.3 Requirements in Detail

Based on our example project Travis CI we will now examine the requirements more closely, reveal potential problems, and introduce their solutions. Figure 4.2 illustrates the architecture's excerpt which we focused on.

Storing the Dependency Relations

Since the module `travis-api` depends, next to others, on `travis-core` and `travis-lib`, the developer will declare these relationships on the online settings page under the heading *Dependencies*¹⁹ (see Figure 4.5). It was a conscious decision not to place the dependency declaration into the `.travis.yml`, because its contents are supposed to describe *how* a build has to happen, not *when*. If the switch is set to *implicit*, the user can write the path of the setup shell script into the *Script* field (see Figure 4.5).

Travis will thus need a new `dependencies` database table which can, together with example entries, be viewed in Figure 4.12, while the corresponding database migration is shown in Listing 4.2, both in Section 4.5.1.

¹⁹Our solution does not take branches into account, because typically dependants make use of the version in the default branch. Still, branches could also be considered in the future which is discussed in Section 4.8.

After the user has clicked on *add*, Travis Web will send the names of the dependency and the dependant, as well as the dependency type, and, if implicit, the path to the custom setup script, to Travis API for it to store the new relation in the database.

Dependency Cycles

The dependency graph of our example is cyclical (cf. Figure 4.2), which would create never-ending build sequences, visualised in Figure 4.6. Travis will hence need to keep track of already tested repositories within each build sequence. Thus when a push or pull request to `travis-lib` has set off a Travis CI test, DependencyManager will, after the build is done, look through the dependency table and add the repository to the list of those that have already been tested within this sequence. Additionally, such a list is necessary for a dynamic presentation of the build sequence on the website, which will be discussed in the *Workflow* passage of this Section.

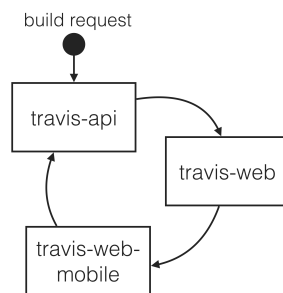


Figure 4.6: The never-ending build sequence resulting from the cycle in the dependency graph. The arrows represent the triggering of its target’s build.

Transitive Dependencies

Our example includes two instances of an additional type of dependency, which in this work we will call *transitive* dependency. This type of dependency can be best illustrated by regarding the modules `travis-api`, `travis-core` and `travis-lib`. Since both `travis-api` and `travis-core` depend on `travis-lib`, a build of the latter should yield each of the dependants to also run a test. However, `travis-core` is another dependency of `travis-api`, so it would be tested again after the former has finished.

This double testing is obviously unnecessary and should be avoided, which is why the design should recognise this type of structure and then automatically skip the builds. This is made possible by the column `skip` in the database table. Whenever an entry to the table is created or deleted, DependencyManager will run an algorithm that traverses the dependency graph of which the new entry is a part of, identifies the transitive dependencies, and updates the value of `skip` for all of the graph’s relations. Circular dependencies are taken into account, otherwise all of the circle’s members would be skipped. Section 4.5.2 contains a code draft of the algorithm.

A different solution would have been to just delete the skippable entries. However, deletion would only be appropriate if the dependency graphs forbid to have cycles and if they would never change again, because alterations can always remove the transitivity, making it incorrect to skip the relation. The resulting build sequence should equal the one depicted in Figure 4.7.

Running a Build Sequence

Now we will explain in detail how build sequences are created and executed. Its core component is our newly proposed `DependencyManager` module. Most of the work will be done there, resulting in a solution for which changes have to be made to only three of the already existing modules, namely `Hub`, `Gatekeeper`, and `Travis Build`.

When a repository is tested by Travis, it has been triggered by an event like a push or a pull request on GitHub.²⁰ A build sequence is hence started exactly the same way.

The original event causes the Listener to send a build request to Gatekeeper, resulting in a build executed by a Travis Worker. When a build has finished, Hub then sends a message to the new `DependencyManager` module informing it about the finished build.

By checking the type of the event that caused the build, `DependencyManager` then inspects whether the finished build is already part of a build sequence and if not, starts a new one by generating a unique ID. Next, it looks into the dependency-table in the database for dependants of the build's repository that are not supposed to be skipped and that have not yet been built within this sequence. If the resulting list of repositories is empty, `DependencyManager` ends the algorithm, discarding any newly generated build sequence ID. If this list is on the other hand not empty, it will proceed as follows. If the currently handled build is the very first one in the sequence, the new ID will have to be added to the build's database entry. For each of the dependants it creates a build request of the through new dependency type, which holds the sequence's ID. After this the algorithm ends.

Gatekeeper creates the corresponding builds and adds the sequence's ID to each build's database entry. If it was a dependent build, Travis Build will, depending on the dependency type, inject the setup script discussed in 4.4.2 into the build scripts for the Workers.²¹ Now the build can be run and the modules' behavior will repeat.

Pull Requests

Next to just pushing changes and seeing whether this has caused any unforeseen damages, it could also be important to know if the changes from merging a pull request would bring about any repercussions for dependant repositories.

²⁰A cron event as introduced in chapter 3 should not trigger build sequences, since no changes have been made to the code. The same goes for builds triggered through API requests.

²¹Since Travis Build does not have a connection to the database, a way for it to get the scripts will have to be added.



Figure 4.7: A mockup of the build sequence view on the `travis-ci.org` website

When making a pull request to a branch of any repository that is tested on Travis, GitHub will check whether a merge would have any conflicts, and only if not will the Travis Listener create a build request. The Travis Worker would clone it via Git from a link provided by GitHub that points to the repository's version in which the pull request is merged and then run the tests as usual.

The behavior of the build sequence triggered by a pull request will be the same as in the description above, with the only exception that Travis Build will alter the setup scripts for dependants to clone the dependency's version of the merged pull request instead of the usual default branch, like in a usual pull request build. The supplementary info on the status of the dependants' build should then be included in the pull request view on the GitHub page, as shown in Figure 4.8.

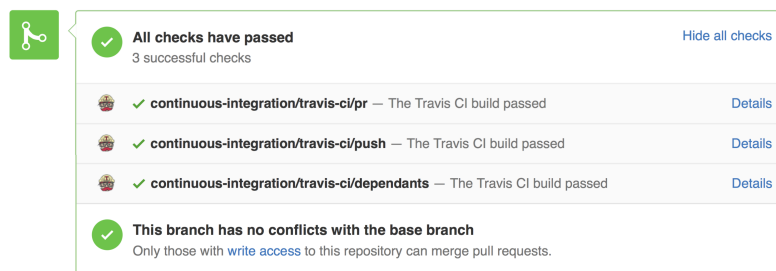


Figure 4.8: Pull request info mockup for GitHub with additional check for dependants

Presenting the Results

After all the tests have run, the developers will have to be notified about the outcomes of the build sequence. As usual, users can set up notifications via email, Slack, *etc.* in their `.travis.yml` and will learn about failing tests of their repositories' dependencies and dependants accordingly. Email notifications should contain the same visual summary of the sequence as in the build sequence view on the Travis CI website (see Figure 4.7) to which it should also link. This view is further discussed in the *Workflow* passage of this Section.

We additionally propose two new status badges,²² shown in Figure 4.9, that will be able to represent the following information:

1. Whether the repository is still compatible with the latest changes pushed to its dependencies
2. Whether the repository's dependants are still working after the changes from the latest push

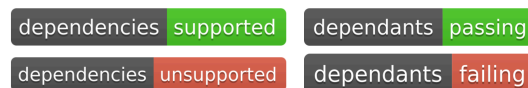


Figure 4.9: The new status badges for dependencies and dependants

They can, just like typical badges on GitHub, be added to the `README.md` file, so that their status can be seen immediately on arrival on the GitHub page. The first badge can be used alongside the typical Travis build status badge, its state will refer to the latest build caused by push or dependency events, while the normal one will only point towards the latest build caused by a push event. This procures that, if the dependencies' changes are released, the lower the frequency of push events, the less likely it is that these two badges will have the same state.

The second badge is available to show the developer, whether their latest push has had any (probably unintended) impact on other repositories or not. If the former is the case, and they wanted to know the specifics to start fixing the problem, they would then be able to click on the badge and be redirected to the build sequence view on the `travis-ci.org` page (see Figure 4.7).

A third important information would be, whether the repository's dependants break because of a change to itself, or to a dependency. This does however not require an own badge, since this is already revealed by the combination of the other two.

The Workflow

We will now consider the following scenario. `travis-lib`'s latest test has failed and a developer is now fixing the issues. The changes that caused the failed build do

²²The badges were made using the service of <http://shields.io>.

not concern the dependants, so their status is still *passing*. The developer now pushes their bugfixes to GitHub, which causes the build sequence to start. After the successful build, `travis-core` is tested, then in turn `travis-api` and `travis-sidekiq`, and the former's passing of its tests triggers the test execution for `travis-web`, which ends up failing, as does its dependant `travis-web-mobile`, which we will ignore in this scenario. The owners of all the tested modules will thereupon be notified, if their `.travis.yml` says so.

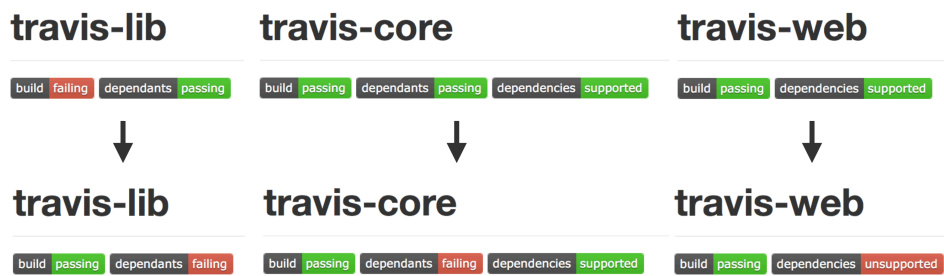


Figure 4.10: The badge status changes to the repositories in the scenario described in Section 4.4.3. `travis-api` has been omitted, since its badges would be equal to the ones of `travis-core`.

The developer would be able to observe the badge changes presented in Figure 4.10 and the desired workflow that could ensue would look as follows:

Because the work had only been done on `travis-lib`, the developer will probably look at this repository's badges, which already hold all the information they need. Since the dependants-badge shows the status *failing*, the developer is then going to go to the *build sequence view* on the website by clicking the badge. This view shows the build sequence tree with the original event, be it a push or a pull request, that triggered the first test and caused the succession to start, on top. There they will see what is shown in Figure 4.7 and are able to identify that it was `travis-web` that has failed. By clicking on its box, they will then be redirected to its build log, where they can go through the failed tests, maybe fix them and either push the changes or create a pull request. If the subsequent build succeeds, the dependencies-badge status will change to *supported*, since it always refers to the latest build not caused by a pull request event. The dependants-badges of the other three repositories will also return to *passing* since they only refer to the AND'ing of the dependencies-badge status of the dependants.

Removing a Dependency Relation

An uncomplicated workflow equal or close to the one just described is the main goal of our design. But another far simpler possibility of how developers can sometimes solve the loss of support for an updated dependency should still be mentioned. Unstable explicit dependencies can always be changed to stable ones by adding the last version number that had been supported to their declaration. Afterwards of course

should the declaration of the dependency be removed from the dependant's settings on Travis, since the dependency feature is only useful for unstable dependencies.

Such a removal of a dependency alters the graph which it was part of. Hence `DependencyManager` must run the transitivity algorithm again. It checks whether the removed relation was skippable (in which case the removal did not have any further effects) and if not, update the skip flag of all repositories that in a build sequence would come before the dependency and those that would be built after the dependant. The algorithm is shown in Listing 4.5 in Section 4.5.2.

If a repository that is part of a dependency relation is deleted on GitHub or deactivated on Travis, its entries will probably also have to be removed from the `dependencies` table, with the just described behavior in consequence.

External Dependencies

As of now we have mainly focused on dependency management as it already exists in tools like Jenkins, which is already a helpful new feature for Travis CI. But it is the combination with the repository network of GitHub that opens up the dependency graphs to include external software, which are actually the cause for unexpected build failures. And since our design does not distinguish between external or internal dependencies, it can improve the users' confidence when employing unstable external dependencies and hence encourage the use of bleeding edge software.

However, leaving it open to users to establish dependency relations between their and any other project on Travis CI would potentially add immense amounts of new builds and also create very confusing and overly deep build sequence trees. To solve this problem without entirely taking away the possibility to include external repositories in a dependency graph, we propose *dependency requests*. When a user adds a dependency that is not part of one of their GitHub organizations, Travis will, before making the database entry, send a request to the dependency's owner to accept the dependant.

4.5 Implementation

In this Section we present implementation details of the proposed design in Section 4.4. We will only focus on adapting the database and API to the data needed for the design as well as our newly proposed module addition to the Travis infrastructure, *DependencyManager*. Other parts of a full implementation, such as the communication protocols between `DependencyManager` and `Gatekeeper` or API, the user interface for Travis Web, or the new setup scripts for Travis Build are details that still need clear specifications, which would at some point be modelled after the eventual use cases.

The code presented here is not part of an implementation in Travis CI. While the code in Section 4.5.1 has actually been tested to be compatible with Travis CI²³ that

²³Tried with `travis-api`'s and `travis-migrations`'s state on June 10, 2016.

dependant_id	dependency_id	implicit	script	skip
2	1	FALSE	NULL	FALSE
3	2	FALSE	NULL	FALSE
4	1	FALSE	NULL	TRUE
4	2	FALSE	NULL	FALSE
4	6	FALSE	NULL	FALSE
4	7	FALSE	NULL	FALSE
5	1	FALSE	NULL	TRUE
5	2	FALSE	NULL	TRUE
5	4	TRUE	"/scripts/deploy.sh"	FALSE
6	5	FALSE	NULL	FALSE

Figure 4.11: The new database table for dependencies, referencing the tables from Figure 4.12

in Section 4.5.2 is just a code draft and does not reference any other existing code in the Travis architecture.

4.5.1 Data Storage

The first step to implementing our solution would be to add the new database table dependencies and to extend the `build` table, by adding a database migration to `travis-migrations` (see Listing 4.2). The `dependency_id` and `dependant_id` entries should be foreign keys to `repository_id`'s from the `repository`-table. An entry in the `script` column is either `NULL` or a string describing the path to the shell-script, starting at the root of the dependant's repository folder, while the entries in `skip` are booleans.

To keep a list of the repositories already tested within a build sequence, each new sequence gets an own ID and whenever a build that is part of it finishes, this ID is added to that build's entry in the `build`-table, for which that table will need a new `build_sequence_id` column (see Figure 4.12). Hence the resulting tree is built up dynamically and it would be possible to visualise this on the website, in contrast to storing the sequence all at once when it has finished. A more detailed description of when and by whom everything is stored can be found in this Section's *Running a Build Sequence* passage.

In our design, users create dependency relations on the website, which communicates with API via restful HTTP requests. API would hence get additional routes (see Listing 4.3). Then it would also be possible to extend the Travis CLI with the dependency functionality. API would furthermore need a model to represent the new Dependency type (see Listing 4.4).

The `Build` model on the other hand would need no changes, since the new `build_sequence_id` is just an attribute, no foreign key, and attributes do not have to be specified in ActiveRecord models.

Listing 4.2: The migration to the database adding the dependencies table and extending the build table

```
1 class AddDependencies < ActiveRecord::Migration
2
3   def up
4     create_table :dependencies do |t|
5       t.references :dependency
6       t.references :dependant
7       t.boolean   :implicit
8       t.string    :script
9       t.boolean   :skip
10    end
11  end
12
13  def change
14    add_column :builds, :build_sequence_id, :integer
15  end
16
17 end
```

Listing 4.3: The changes needed to be made to the resource :repository entry in the routes.rb file in the travis-api repository

```
126   resource :dependencies do
127     get :find,      '/dependencies'
128     post :create,  '/dependencies'
129     delete :delete, '/dependencies'
130
131     get :find,      '/dependants'
132   end
```

Listing 4.4: The model of the dependencies table

```
1 module Travis::API::V3
2   class Models::Dependency < Model
3     belongs_to :repository, as: :dependency
4     belongs_to :repository, as: :dependant
5   end
6 end
```

repository_id	name	build_id	repository_id	build_sequence_id
1	travis-lib	2418	1	1
2	travis-core	2419	2	1
3	travis-sidekiq	2420	4	1
4	travis-api	2421	3	1
5	travis-web	2422	5	1
6	travis-web-mobile	2423	6	1
7	sinatra			

Figure 4.12: Parts of the repository and the build table in the database, with the build_sequence_id column added to the latter. The fictitious values represent the structure defined in Figure 4.2 and correspond to the build sequence in Figure 4.7, both in Section 4.4.3

4.5.2 DependencyManager

The DependencyManager introduced in Section 4.4 would need to be an independent module which could for example be deployed to Heroku, just like Travis Hub. It needs communication channels to API, Hub, and Gatekeeper, which could potentially be established using Sidekiq (see Figure 4.13). Its two main functionalities are taking care of transitivities in dependency graphs and making build requests for dependants when appropriate.

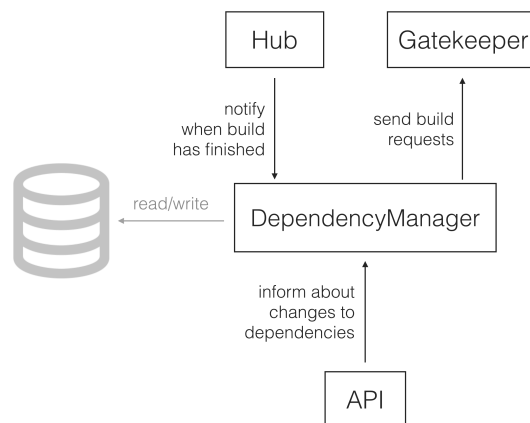


Figure 4.13: Communication channels of DependencyManager

Transitivity Algorithms

After having added or removed a dependency, the Travis API will then notify the DependencyManager by sending it the type of change that was made, as well as the two repositories in the relation. DependencyManager then runs one of the transitivity algorithms, depending on the situation.

When adding a dependency relation, the algorithm goes through all the relations in the graph that the new one is part of and see, whether some of them have become transitive, which it then marks as skippable. Because we are only adding a relation, it is impossible that an unskippable relation becomes skippable.

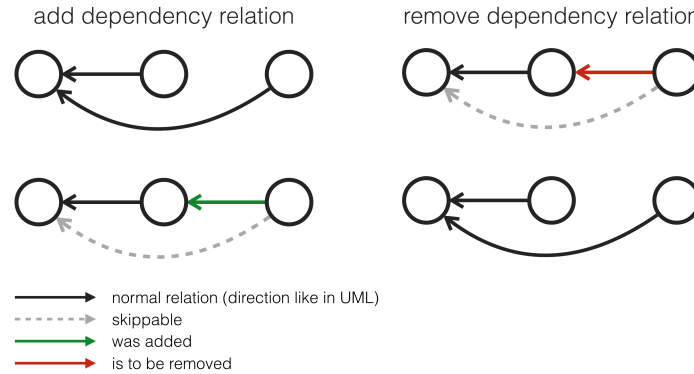


Figure 4.14: Examples for the results of the transitivity algorithms, the upper graphs being in the state before the operation, those below afterwards

When removing a dependency relation, the algorithm first checks whether the removed one was skippable, in which case it can terminate, and if not, it checks whether parts of its corresponding graph needs to be updated, since it might have been cut in half. Because we are only removing a relation, it is only possible that a skippable relation becomes unskippable.

Cyclical dependency paths are also taken care of, so that when creating a cycle not all of its elements are skipped.

The pseudo code in Listing 4.5 shows a possible implementation, while Figure 4.14 visualises the changes based on simple examples.

Making Build Requests

Whenever Hub is notified by a Worker about a finished build, it forwards this information to DependencyManager. Listing 4.6 shows the ensuing behavior through the method `handleFinishedBuild(build)`, which entails handling the start of a build sequence correctly (Lines 3–5, 12–14), finding dependants that need to be built (Lines 20 ff.), and making build requests for them (Line 16).

Listing 4.5: A code draft for DependencyManager's transitivity algorithms

```

1 def addRelation(relation)
2   #iterate over all possible relations that could now be skipped
3   for relevantRelations(relation) do |each|
4     if each != relation
5       #each is not part of a circle
6       if !pathExistsFromTo(each.dependency, each.dependency)
7         each.skip = true
8       end
9     end
10  end
11  #finally check new relation
12  updateSkipAttr(relation)
13 end
14
15 def removeRelation(relation)
16  #if the deleted relation was skipped, nothing happens
17  if !relation.skip
18    for relevantRelations(relation) do |each|
19      #only those that are already skipped might not be skipped anymore
20      if each.skip
21        updateSkipAttr(each)
22      end
23    end
24  end
25 end
26
27 def updateSkipAttr(relation)
28  for pathsFromTo(relation.dependency, relation.dependant) do |path|
29    if !path.relations.includes(relation)
30      relation.skip = true
31      return
32    end
33  end
34  relation.skip = false
35 end
36
37 def relevantRelations(relation)
38  #returns all relations that are included in paths ending in
39  #  relation.dependency or starting from relation.dependant
40  relations = []
41  for pathsTo(relation.dependency).collect(nodes).flatten do |nodeAbove|
42    for pathsFrom(relation.dependant).collect(nodes).flatten do |nodeBelow|
43      relations.push(getRelationIfExistent(nodeAbove, nodeBelow))
44    end
45  end
46  #remove all nils
47  return relations.compact
48 end

```

Listing 4.6: A code draft for DependencyManager's handling of finished builds

```
1
2 def handleFinishedBuild(build)
3   if build.event_type != 'dependency'
4     sequenceID = self.getNextSequenceID()
5   else
6     sequenceID = build.sequenceID
7   end
8   dependants = self.getDependantsToBeBuilt(build.repository, sequenceID)
9   if dependants.empty?
10    return
11  end
12  if build.event_type != 'dependency'
13    self.storeSequenceIDtoBuildInDatabase(build, sequenceID)
14  end
15  dependants.each do |dependant|
16    self.createBuildRequestFor(dependant)
17  end
18 end
19
20 def getDependantsToBeBuilt(repository, sequenceID)
21  dependants = []
22  repository.dependants.each do |dependant|
23    if !dependant.skip && !self.alreadyBuilt(dependant.id, sequenceID)
24      dependants.push(dependant)
25    end
26  end
27  return dependants
28 end
```

4.6 Evaluation

Our approach is a trade-off between complexity and usefulness. We left out features like pull request testing for implicit dependencies to reduce complexity, but let users create dependency graphs containing multiple dependency relations for build sequences. Instead we could have constrained these graphs to only one relation. This way most of the problems discussed in this chapter would not arise. This way it would take a lot less effort to implement the feature, but also yield far less information for the users, drastically reducing the usefulness. This also leads to the question of how valuable the generated information actually is.

Build sequences like the one shown in Figure 4.7 are in some ways problematic because there can be various different explanations for what actually caused the build to fail. It could be that the changes to `travis-lib` have caused a waterfall of behavior changes to `travis-core`, `travis-api`, or both, so that `travis-web` could not use them anymore, or the direct relations of `travis-lib` and `travis-core`, which are both not visible in the build sequence view, but still defined in the settings could lie at the root of the problem. Even though it is interesting that something has been broken, the multitude of possible explanations does not at all help the user to know what to do and where to look for the problems' cause. The feature's focus should therefore be well defined. Is the main goal to just inform the user that something has broken, or should the user also be supported in finding the cause? If the latter is the goal, then the solution should be extended, for example by also making it possible to present the transitive dependencies in the build sequence view. Otherwise the proposed solution should suffice.

Just like architectures with multiple inheritance in object-oriented design, dependency graphs are susceptible to the *diamond problem* [9, 33]. If such a constellation occurs, our solution will behave nondeterministically. If a build sequence based on a dependency graph with the diamond dependency $(B \succ A) \wedge (C \succ A) \wedge (D \succ B) \wedge (D \succ C)$ was started by A , D would only be built once, triggered by whichever of B or C finishes first, since `DependencyManager` allows a repository to be tested only once per sequence. The slower one would hence not be considered in the build script, which could potentially be a problem for some users.

If implemented, the dependent builds feature would have a lot of impact on Travis. Not only might already existing users test their software more extensively, but it could also attract many new developers to start using the service, resulting in large amounts of additional builds.

But since Travis CI is a commercial company, feasibility is an important factor when deciding on adding a feature or not. The financial costs caused by its development, as well as all the additional builds should reflect the added benefits for the users. Thus any implementation should have preceding cost impact calculations. If these would come to show that for it to be feasible, the feature would need to somehow be constrained, the following ideas could be considered.

- The design could have dependency creation limits for each repository, user, or organization.

- Sequences could only be triggered if a push contains commits with a certain tag exactly for that (which could be either defined by the user or Travis).
- Build sequences could stop after a certain depth.
- The whole feature or parts of it could be only available to paying costumers.
- External dependencies could be left out/paying customer only.

4.7 Related Work

The focus of this Section is the examination of the topic of dependency resolution in package management, the plethora of different CI tools, and their intersection. We will look at Jenkins CI, its solution for build sequences along dependency trees, and its ideas and constraints that have laid the groundwork for this chapter.

To help end users with the complex²⁴ task of installing an application, it is nowadays common for operating systems to include some form of *package management*. A *package* is usually an archive file with the software itself and metadata such as the version number, the vendor, and a list of dependencies. When one such package is to be installed, a package manager such as Debian's `dpkg`, will i.a. go through that dependency list, compare it to its list of already installed software and install anything from the list of 43,000 packages²⁵ supported by Debian that is needed but missing. This dependency resolution is often a nontrivial task, since dependency relations can be e.g. conflicting, circular, or even both [43]. In addition, Debian packages can specify their dependency relations in a finely tuned way with control fields like `Pre-Depends`, `Suggests`, and `Enhances`, instead of just `Depends`,²⁶ which adds another layer of complexity.

This process of installing packages is analogous to building software from source code to executable artefacts, which is why build automation utilities like `make` and `Gradle` are needed to reduce effort and time spent on a build. They are often a cornerstone of continuous integration and delivery workflows [19].

Continuous Integration is a "*practice where members of a team integrate their work frequently*" instead of merging large change sets after long periods of time [19]. To complement and secure this practice, tests and builds are automated using CI tools like `CruiseControl`²⁷ and `Jenkins`.²⁸ They can be triggered by any custom event, such as a push to an integrated version control system, a periodical script, or finished builds of other packages within an architecture.

²⁴<http://ianmurdock.debian.net/index.html%3Fp=437.html>
(visited on 2016-06-30).

²⁵<https://www.debian.org/intro/about> (visited on 2016-06-30).

²⁶<https://www.debian.org/doc/debian-policy/ch-relationships.html>
(visited on 2016-06-30).

²⁷<http://cruisecontrol.sourceforge.net> (visited on 2016-06-30).

²⁸<https://jenkins.io> (visited on 2016-06-30).

Jenkins users can create post-build actions through which they can create build sequences comparable to the ones in our solution [18], except they have the additional possibility to set actions that depend on the build's outcome. The Build Pipeline Plugin and the Downstream Buildview Plugin²⁹ can be used to organize build pipelines along dependency graphs and to get a visual overview of their structure and outcomes [28].

The build automation tools used by the CI tools can handle dependencies and integrate them into the testing process which is an important part of our solution. However, they cannot be made part of the build pipeline, meaning that the build triggers are limited to the context of the CI tool setup. Furthermore, most CI tools are constrained to only one or a few programming languages and operating systems, making it difficult, if not impossible, to create complete build pipelines for architectures in which modules are written in different languages³⁰.

These shortcomings are the focus of our solution. To resolve them, it combines the dependency management of CI tools like Jenkins with Travis CI's possibility to test on a big range of technologies and its large community of interdependent software packages. The hosted CI service Shippable already supports build pipelines, but constrained to the user's repositories³¹. Hosted CI services are a relatively new phenomenon, and their feature lists are still growing rapidly. It is thus likely that build pipelines will eventually become a common feature for them, but as of now they are not.

4.8 Future Work

Software dependencies are a large and complex problem field [45] which has to be tackled on many different levels in all kinds of contexts, since there will probably never be a *one size fits all* solution. Our proposed feature addition is thus by no means the last word on the subject of unstable dependencies and leaves room for extensions and optimisations.

Jenkins gives its users the ability to specify whether to build a dependant based on the outcome of the dependency's build. This would mark a desirable feature addition to our solution.

Another improvement would be to allow the specification of branches for the dependencies, since our solution only supports the default branch. It could especially be useful within organizations when working on different production stages which are all represented by a corresponding branch. Then each stage could have its own dependency graph and its own setup scripts.

²⁹<https://wiki.jenkins-ci.org/display/JENKINS/Downstream+buildview+plugin> (visited on 2016-06-30).

³⁰A comparison of different CI tools can be viewed at https://en.wikipedia.org/wiki/Comparison_of_continuous_integration_software (visited on 2016-06-30).

³¹http://docs.shippable.com/pipelines_configure/ (visited on 2016-06-30).

Travis could also start analysing the repositories' explicit dependency declarations by actually parsing the code (e.g. the Gemfile for Ruby projects), see whether some of them are also on GitHub and tested on Travis, and then prompt the user asking whether they also want to add these dependencies.

Furthermore, when the overview page discussed in chapter 5 is added to the Travis CI website, the number, status *etc.* of the repository's dependencies and dependants could be added.

When specifying and designing the minute details of the setup script creation for Travis Build, it would probably be helpful to study Maven's *transitive dependencies* feature which automatically detects dependencies of dependencies *etc.* and installs them appropriately³²

Since the setup script for each new dependant equals the one of its dependency with only some other script concatenated to the end, an optimisation could be added to the build sequence design, because this means that in case a setup script fails, the whole build will fail, and all consequent builds in the sequence will do so too. If it were possible to find out that the build has failed because of the setup script, the rest of the build sequence could be added to the database with a failed state without actually running any of the other builds.

And of course, the proposed feature still needs to be implemented.

4.9 Conclusion

In this chapter we have proposed and discussed our design of an additional feature for the hosted continuous integration service Travis CI that would extend it with dependent builds for both explicit and implicit dependencies. Our solution would help users solve the problem of unstable dependencies and generate a workflow to identify issues within a modular architecture and with external dependencies. We have discussed the ways of testing the different types of dependencies, introduced build sequences, and ways of presenting the generated information, including our newly proposed status badges. Implementation details were presented and the trade-off between complexity and usefulness was evaluated. We gave an overview of technologies in the domain of dependency management, continuous integration, as well as their intersection, and gave an outlook on potential future work on our solution.

If dependent builds were to become part of the list of features accessible to every open source project using Travis for free, the open source community could potentially benefit greatly. They would stay in control of their internal as well as external dependencies and might get encouraged to depend on bleeding edge software more often.

³²<https://maven.apache.org/guides/introduction/introduction-to-dependency-mechanism.html> (visited on 2016-06-27).

5 Visualizing Build Data in Continuous Integration Services

Travis is a continuous integration service that runs tests and collects their data. For developers it is hard to get the big picture from this collected raw data. We analyze which data is useful to collect and display in graphical form to provide developers helpful insights into the state of their repositories. We develop an overview page which contains statistics like build duration over time as well as the recent build history and implement it into Travis' website. With this overview development teams can get valuable metrics of their repositories.

5.1 Introduction

Continuous integration is a common practice where code changes are tested, merged, and deployed several times a day. Hosted continuous integration services like Travis provide testing and deployment in the cloud. While running the tests, these services collect data such as their status and duration about builds.

The Travis website spreads this data over different pages, therefore it is hard to see the big picture. With a comparison of the build duration over time developers can notice if their software performance decreases. But without overview it is difficult for the developers to retrieve any such valuable information from the build data.

In this chapter we are going to analyze what data is useful to collect in the build process to give development teams insights which can lead to better software. We focus on build meta data from Travis and do not connect them with data from different sources. Furthermore we look at different visualizations of the build data. This includes revealing problems of external services which already offer visualization for repositories on Travis.

We develop a new overview page for repositories containing diagrams and statistics. These representations include a diagram with build duration over time, the recent build history, and the number of days since the last failed build.

5.2 State of the Art

The Travis website has no specific focus on visualization. Instead it is built for easy and fast retrieval of information about the latest builds. Therefore, based on the Travis API, independent developers programmed other tools for visualization. In this chapter we are going to have a look at relevant parts of the Travis website as

well as some tools developed by third parties. As it is necessary to know the data before trying to visualize them we are going to start out by giving an overview.

5.2.1 Build Data

Travis collects the following data for each build:

Status The status the build is in. This can be success, errored, failed, cancelled, or a status which indicates that the build is not finished yet, like running or booting.

Type The Event which caused the build. This indicates whether a build was caused by a git push (push), an opened pull request (pull_request), API call (api), or cron job (cron).

Start time Timestamp of the start of the build.

End time Timestamp of the end of the build.

Elapsed time On Travis a build can consist of multiple jobs. In this case, Travis can execute more than one job at the same time. The time from the start of the first until the end of the last job is the elapsed time.

Total time The sum of the duration of all jobs. If the build has only one job this is equal to the elapsed time.

Travis adds the following information for each build:

Number This number is counted per repository. It is incremented for each build in this repository.

ID The ID is unique for all builds in a Travis instance¹ It identifies a build without needing to know the repository.

Each test on Travis does a checkout of a git commit. Travis uses the following data from this commit:

Branch The branch the commit is on. In case of a pull_request build this is the branch the commits should be merged into.

Author The person who wrote the code changes. If existing, the Gravatar² of this person is shown next to the name.

Committer The person who committed the code changes. If existing, the Gravatar of this person is shown next to the name.

¹Currently the .org and .com services from Travis are two separate instances.

²A Gravatar is a unified profile image which is used on different websites, see <http://en.gravatar.com> (visited on 2016-07-06).

Commit message The message of the commit. For `pull_request` builds the title of the pull request is used instead.

Commit hash The unique hash of the git commit. For `pull_request` builds a merge commit is created.

5.2.2 Travis Website

To structure the information about a repository on the Travis website it is divided into different pages. These pages are displayed as the following tabs: Current, Branches, Build History, and Pull Requests (see Figure 5.1).

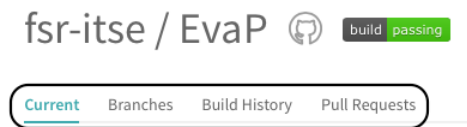


Figure 5.1: Navigation for a repository on Travis

Current

When opening a repository on Travis, for example by clicking on a Travis badge³ the Current tab is shown. This page displays the last build of this repository as shown in Figure 5.2. The information displayed on a page with a single build above the log is explained later.

Branches

On the tab Branches is a branch row (1) for each branch shown (see Figure 5.3). On the left side of this row the status of the last build is indicated by a color (2) and an icon (3). Next to this are the branch (4) and number of builds on this branch (5). For the last build the type as icon, number, and status (6), finished date (7), commit hash with link to GitHub (8), and committer (9) are additionally shown. On the right side of the row the status of the last 5 builds is shown, with (10) being the last build. When hovering over a build status, the website reveals the build number for this build, too.

³Travis badges are small images showing the build status of one selected branch and are often included in the readme file of a repository.

5 Visualizing Build Data in Continuous Integration Services

fsr-itse / EvaP build passing

Current Branches Build History Pull Requests More options

✓ master Merge pull request #805 from janno42/fix_802 → #1695 passed

Fix adding multiple questions at once Elapsed time 2 min 50 sec

Commit afa8d0a about 20 hours ago

Compare 52f4b7e..afa8d0a

Johannes Wolf authored and committed

```
1 Using worker: worker-linux-docker-650df49d.prod.travis-ci.org:travis-linux-5
2
3 Build system information (system:info)
67
68 $ export DEBIAN_FRONTEND=noninteractive (fix:CFE:2015-7547) (0:36s)
69 $ git clone --depth=50 --branch=master https://github.com/fsr-itse/EvaP.git fsr-itse/EvaP (git:checkout) (0:22s)
70 $ git submodule update --init --recursive (git:submodule) (0:22s)
162
163 This job is running on container-based infrastructure, which does not allow use of 'sudo', setuid and setgid executables.
164 If you require sudo, add 'sudo: required' to your .travis.yml
165 See https://docs.travis-ci.com/user/workers/container-based-infrastructure/ for details.
166 $ source ~/.virtualenv/python3.4/bin/activate (0:00s)
167
168 Setting up build cache (cache:1)
181 $ python --version
182 Python 3.4.2
183 $ pip --version
184 pip 6.0.7 from /home/travis/virtualenv/python3.4.2/lib/python3.4/site-packages (python 3.4)
185 $ npm install -g less (install:1) (4:56s)
186 $ npm install -r requirements.txt (install:2) (29:51s)
187 $ pip install coveralls (install:3) (41:03s)
188 $ coverage run manage.py test evap.evaluation evap.staff evap.contributor evap.results evap.student evap.grades evap.rewards evap.evaluation.tests.test_misc.TestDataTest.load_test_data (0:00s)
301 Creating test database for alias 'default'...
302 .....
303 Ran 156 tests in 52.275s
304 .....
305 OK
306 Destroying test database for alias 'default'...
307
308 The command "coverage run manage.py test evap.evaluation evap.staff evap.contributor evap.results evap.student evap.grades evap.rewards evap.evaluation.tests.test_misc.TestDataTest.load_test_data" exited with 0.
312 store build cache (cache:2)
332 $ coveralls (after_success) (2:39s)
338
```

Figure 5.2: Current tab, which is the default view of a repository on Travis

hpi-swa / smalltalkCI build passing

Current Branches Build History Pull Requests More options

Default Branch

3	4	6	8	10
✓ master	→ #500 passed	5fd4c78	✓	✓
122 builds	3 days ago	Fabio Niephaus	✓	✓

Active Branches

✓ dev	→ #498 passed	cdfa4ba	✓	○	✗	○	○
179 builds	6 days ago	Fabio Niephaus					
✗ issue/119	→ #398 failed	203e27d	✗				
1 builds	2 months ago	Fabio Niephaus					

Figure 5.3: Overview of the last 5 builds of each branch on the Branches tab

Build History

The Build History tab displays a list of all but `pull_request` builds ordered by the creation⁴ as shown in Figure 5.4. For each build the status (1, 2) and branch (3) are shown like in a branch row. Next to this the author (4) is displayed. Additionally to the data shown on the Branches tab the commit message is displayed in the center (5). On the right side the build type as icon, number, and status (6), commit hash with link to GitHub (7), total build time (8), and finished date (9) are shown.

hpi-swa / smalltalkCI build passing

Current Branches **Build History** Pull Requests More options

1	2	3	4	5	6	7	8	9
✓	master	Fabio Niephaus	Custom STON can be a relative path; fixes #147	→ #500 passed	5fd4c78	27 min 48 sec	3 days ago	
✓	master	Fabio Niephaus	Improve compatibility to GemStone #144	→ #499 passed	cdfa4ba	33 min 32 sec	6 days ago	
✓	dev	Fabio Niephaus	Improve compatibility to GemStone #144	→ #498 passed	cdfa4ba	32 min 21 sec	6 days ago	
✓	master	Fabio Niephaus	Make sure to prefer STON config provided as cmd	→ #497 passed	69a8ff9	54 min 15 sec	7 days ago	
✓	master	Fabio Niephaus	Update and improve bash tests	→ #496 passed	6dde96c	29 min 52 sec	11 days ago	
⊘	dev	Fabio Niephaus	Update and improve bash tests	→ #495 canceled	6dde96c	6 sec	11 days ago	

Figure 5.4: List of the last builds on the Build History tab

Pull Requests

The tab Pull Requests looks like the Build History tab, but only shows the `pull_request` builds, which were excluded from the other one.

Build

On the Current tab, or when selecting a build on one of the other tabs, Travis' website shows a page for a single build. On the top of this page there is a box where all information for this build is displayed, see Figure 5.5. This build header offers information which is also displayed on the other tabs like status (1, 2), branch (3), commit message (4), commit hash with link to GitHub (5) but also new elements like the commit hash of the commit before with a link to GitHub to compare the changes between the last two commits (6). Instead of just the committer or the author both are shown (7). On the right side the known build type, number, and status combination

⁴For more information on how a build is created see chapter 2.

(8) is displayed. For builds with more than one job the header contains the elapsed time (9) in addition to the total time (10). The last element is the time the build was finished (11). For jobs without a matrix the site furthermore displays the build log below this box (as already shown in Figure 5.2).



Figure 5.5: Header of a build with multiple jobs

5.2.3 External Services

Build Duration

Silviu-Cristian Burcă has built a tool⁵ to visualize the duration of the last builds because “the regular Travis CI interface wasn’t very conclusive” [10]. After entering the GitHub slug, the website⁶ of the tool, shown in Figure 5.6, generates two bar charts for the build duration; one for the total time (called duration in the tool) and another for the elapsed time (called wall time). A third bar chart shows the number of builds for all days with builds.

Buildtime Trend

Dieter Adriaenssens has built the tool Buildtime Trend⁷ “to create charts to visualise trends of the build process.” [1]. It offers statistics about builds in a selected time period like days since last fail, build duration by build matrix parameter, and build time per day of week.⁸

⁵The source code is available on GitHub: <https://github.com/scribu/travis-stats> (visited on 2016-06-07).

⁶<http://scribu.github.io/travis-stats/> (visited on 2016-06-07).

⁷The source code is available on GitHub: <https://github.com/buildtimetrend/> (visited on 2016-06-07).

⁸A full list is available at Buildtime Trend as a Service, see <https://buildtimetrend.herokuapp.com> (visited on 2016-06-07).

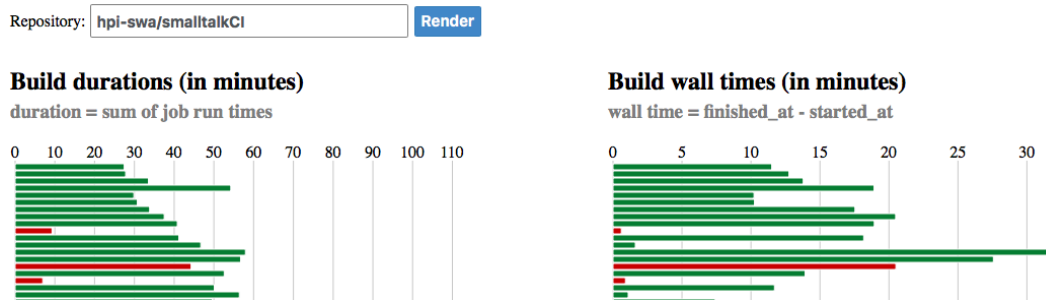


Figure 5.6: Screenshot of the tool to visualize build durations of builds on Travis

Travalizit

Travalizit⁹ is a tool to analyze build data developed by Stefan Judis as part of his bachelor's thesis. It offers the user a customizable dashboard with different diagrams. The possible diagrams include a build duration diagram like the one mentioned above but also diagrams which for example use a connection with data from GitHub to link changed files or the amount of lines edited to the build status [29].

5.3 Problem

In this chapter we are going to examine the insufficient presentation of the data on the Travis website and the shortcomings of external tools introduced in subsection 5.2.3.

5.3.1 Travis Website

On the different pages on the Travis website various attributes of the builds are listed, but only on the build page all available data is displayed. To compare data which is only displayed on this page, for example the elapsed time, it would be necessary to open the pages of all builds to compare.

Even if one wanted to compare data shown on one site problems would arise. On the Branches tab only the status of all displayed builds is shown, so the user cannot for example detect outliers in any other data set. The Build History tab displays more data, but here the builds are not grouped by the branch or event type. For example, to look over the course of time of the build duration it is not appropriate to use builds from different branches or event types because the test configuration or the matrix can be different.

Furthermore, for people it is much easier to interpret graphics instead of raw data, especially if this is spread over different sites [40, p. 5].

⁹The source code is available on GitHub: <https://github.com/stefanjudis/travalizit-app> (visited on 2016-06-13).

After a developer has pushed their code to GitHub during development they often want to check the status of the last build. Currently one can *sometimes* achieve this via the Current tab. The problem is that the last build does not necessarily belong to this developer. Especially when working with in a large team at the same time it often happens that the Current tab shows a build from another developer who pushed more recently.

5.3.2 External Services

External services for visualizing the Travis build have general issues that are not tied to a specific service.

External tools always have the problem of accessibility. First, developers have to look for them and second they need to use different services to access all important visualizations. Third, the communication with Travis through the API is not optimal. Data which need aggregations and calculations cannot be collected optimized in the database – instead all data needs to be loaded via the Travis API and then be processed by the tool. Additionally maintenance of these tools is not guaranteed. If Travis changes the API and tools break it is neither clear when nor if third parties will fix their tools.¹⁰

The build duration tool introduced in section 5.2.3 for example loads about the last 500 builds incrementally via the Travis API [10]. The Travalizit tool does not exist as hosted service, but only as source code. Therefore one has a high barrier to use it, because a possible user has to host it on their own.

5.4 Approach

In this section we are going to present our concept that aims to solve the different problems identified in section 5.3. To avoid the general problems of external services (see subsection 5.3.2) we do not build another external tool but instead introduce some changes to the Travis website itself.

The approach is divided into two sections. The first suggests an overview page for the Travis website and the second a way to collect more data form running tests. The implementations of the overview page is presented in section 5.5 while for the second part only the approach is described and no implementation shown.

5.4.1 Overview Page

We propose to replace the Current tab, which leads to the last build page, with a new overview tab. It could look like the mockup in Figure 5.7. This tab should not be an statistics page, instead we want to build a page, that is the start page if

¹⁰To detect this, the maintainers can use periodic tests as introduced in chapter 3 or automatic testing after changes in dependencies as introduced in chapter 4.

one visits a repository on Travis. Therefore we only want to show selected diagrams and numbers which are understandable at first glance and offer developers helpful insights into their testing and CI usage. To accomplish this we for example focus some diagrams on the default branch even if we can display them separately for each branch. We do not want to show diagrams with data where one cannot interpret anything useful. Hereinafter we explain for each selected statistic why developers benefit from it.

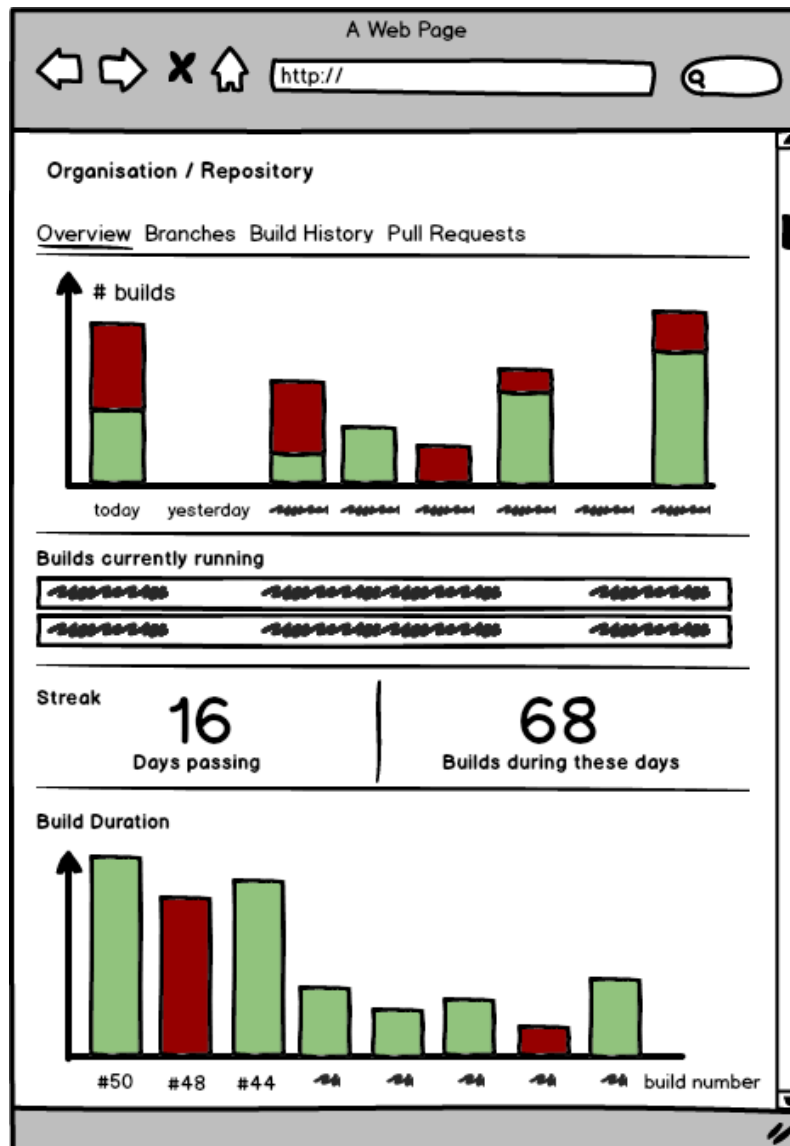


Figure 5.7: Mockup of the new overview page

Build History

This chart shows all finished builds of the last ten days on all branches with their status. An example is shown in the first diagram of Figure 5.7. The diagram depicts the quantity and quality of work in the last days.

By removing the Current tab we avoid leading developers to a build that was not triggered by them. To still have the possibility to navigate to recent builds, we list and link the currently queued and running builds which are not part of the diagram below. With this approach the developer can navigate to their build even if it is not the latest.

Streak

The streak of a repository is the number of days since when builds on the default branch were neither errored, failed nor canceled. If a developer works on a repository and then takes a longer pause this streak will keep growing; therefore we additionally show the numbers of builds in these days. For example if a repository is passing for 200 days but only has seven builds in this timespan one knows it was not actively developed during this time. To reflect the number of builds correctly we only count push builds. We do not count `pull_request` builds because not only the main developers can open pull requests and we do not count cron builds so that a daily cron job for example does not increase the build count every day.

If a development team is testing all pull requests, the default branch should not be broken. So the streak shows if testing techniques are consistently applied. This can increase the trust in open source software projects without failure for a long time while being under active development.

Build Duration

This diagram shows the total time of the last builds with the aid of a bar chart as illustrated by the example in the last diagram in Figure 5.7. It lets developers notice that the builds have become slower since a certain version. In our example the build has gotten slower since build #44. This can indicate code changes which decreased the performance of the software, leading maintainer to go through the code changes which happened to the first slow build, find the relevant part of the code, and fix it.

In this diagram we focus on one branch because a comparison of more than one branch for example can be misleading due to different build configurations. We chose the default branch because developers are usually not interested in foreign development branches.

The color of the bars represents the status of the build, red indicating a fail and green a pass. This is important because builds can fail early and stop without running all tests. We show only these two types because the user can cancel builds at any time, leading to a shorter build duration. So showing canceled builds is misleading in the diagram and running builds do not have a duration yet.

5.4.2 JUnit XML

As shown in subsection 5.2.1 Travis has data about each build, but data about individual tests are not included. However information like the duration of a single test over time or whether failed builds are always caused by the same test can be valuable for developers. We therefore want to add such data to Travis by using JUnit XML output.

Listing 5.1: Schematic description of the JUnit XML format, based on [11]

```

1  <testsuites>           => the aggregated result of all junit test files
2  <testsuite            => the output from a single TestSuite
3  name=""              => name of the test suite
4  errors=""            => number of test cases errored
5  tests=""             => number of test cases
6  failures=""          => number of test cases failed
7  time=""              => time to execute all test cases in this suite
8  >
9  <properties>          => the defined properties at test execution
10 <property              => name/value pair for a single property
11   name=""
12   value=""
13 </>
14 ...
15 </properties>
16 <testcase              => the results from executing a test method
17   classname=""        => name of class containing the test
18   name=""             => name of the test
19   time=""             => time it took to run the test case
20 >
21 <system-out>          => data written to System.out during the test run
22 <system-err>          => data written to System.err during the test run
23 <skipped/>            => test was skipped
24 <failure>             => test failed
25 <error>               => test encountered an error
26 </testcase>
27 ...
28 </testsuite>
29 ...
30 </testsuites>

```

JUnit XML

The JUnit XML file format¹¹ is a format for saving information about individual tests. It is shown schematically in Listing 5.1. The format was originally developed

¹¹There are slightly different definitions; for this chapter we are only using the definition published in [26] which is the version used in Apache Ant 1.8.2.

for Apache Ant, a Java build tool. Hence it has some Java specifics like the package attribute.

Today JUnit XML is widely used. There are tools to convert output from different languages like Ruby,¹² Python,¹³ and Go.¹⁴

Integration into Travis

Since Travis supports 25 different languages [47] and users can also use it to execute bash scripts [46], it is complicated to integrate JUnit XML output for every build. It should thus be optional. In a separate build step the XML should be written to a specific location. To support this for all languages Travis needs to include this in each language specific build script. If users want to use this feature with a custom script they can output the XML.

* * *

The overview page allows developers to get insights into their repositories as soon as they open Travis, for example via a badge on GitHub. On this page they can for example immediately see if the builds of their project have gotten slower and navigate quickly to currently running builds.

5.5 Implementation

Travis uses many microservices. In order to implement the overview page we need to change the API¹⁵ and the Web¹⁶ service. The API is written in Ruby and offers a RESTful interface of requesting all data Travis collects. The Web service is written in Ember.js, a framework based on node.js. The website requests all shown data from the API (chapter 2).

This chapter is divided into an API and a Web section, both containing a part for each introduced diagram.

5.5.1 API

We want to display all statistics on the start page of a repository and therefore users will request them often. The endpoints provided by Travis v3 API¹⁷ are not specific for our suggested overview page. For example to get the streak, one needs to request builds until the first failed, errored, or canceled is found. This is not performant so we decided to add one API endpoint per statistic which returns a

¹²https://github.com/sj26/rspec_junit_formatter (visited on 2016-06-10).

¹³<https://github.com/kyrus/python-junit-xml> (visited on 2016-06-10).

¹⁴<https://github.com/jstemmer/go-junit-report> (visited on 2016-06-10).

¹⁵see <https://github.com/travis-ci/travis-api> (visited on 2016-07-06).

¹⁶see <https://github.com/travis-ci/travis-web> (visited on 2016-07-06).

¹⁷see <https://api.travis-ci.org/v3/> (visited on 2016-05-24).

JSON containing exactly the data needed. With this approach we can do as much calculation as possible within the database which leads to reduced communication and computing effort. All endpoints will look like `https://api.travis-ci.org/v3/repo/{ID|Slug}/overview/{endpoint}` and can be queried with the HTTP method GET. `{ID|Slug}` is either replaced by the repository ID from Travis or the GitHub slug.¹⁸

In the following we are going to show and explain the needed Active Record¹⁹ statements to get the data from the database and define the JSON format returned from each endpoint.

Build History

The endpoint `build_history` returns a JSON as shown in Listing 5.2. Lines 2–4 are meta data which is included in the result of every API call, but not relevant for us. In the `recent_build_history` key (line 5) exists a key for each day of the last ten with at least one build (in this example the keys `2016-05-26`, `2016-05-25`, and `2016-05-24`). Each of these date keys has a key for each build result (passed, failed, and errored). This key contains the number of builds which ended in this result on this day. Keys with a count of zero are left out to reduce the amount of data transferred.

Listing 5.2: JSON returned from the build history overview endpoint

```

1 {
2   "@type": "overview",
3   "@href": "/v3/repo/43487/overview/build_history",
4   "@representation": "standard",
5   "recent_build_history": {
6     "2016-05-26": {
7       "passed": 1
8     },
9     "2016-05-25": {
10      "passed": 1
11    },
12    "2016-05-24": {
13      "passed": 1
14    },
15    ...
16  }
17 }
```

The Active Record statement for this is shown in Listing 5.3. In line 3 the builds of the default branch in the correct repository are selected and in the next line the

¹⁸The GitHub slug consists of the user or organisation name, followed by a / and the name of the repository.

¹⁹Object Relational Mapping system used in the Travis API.

Listing 5.3: Active Record code for the build history endpoint in `lib/travis/api/v3/queries/overview.rb`

```

1 def recent_build_history(repo)
2   Models::Build
3     .where(:repository_id => repo.id, :branch => repo.default_branch_name)
4     .where("finished > ?", Date.today - 9)
5     .group("date_trunc('day', finished)", :state)
6     .count
7 end

```

data range is limited to 10 days (today and the last 9 days). After this all builds are grouped by the state and day which the database extracts from the `finished_at` timestamp. We use the `finished_at` date to exclude builds which are not finished yet and have `finished_at` set to NULL. In line 6 we only request the count for this, so for each date and state we get a number, just as returned in the JSON.

Streak

The JSON returned by the `streak` endpoint contains the number of the days and the number of builds since when the last build on the default branch was broken, see Listing 5.4.

Listing 5.4: JSON returned from the streak endpoint

```

1 {
2   "@type": "overview",
3   "@href": "/v3/repo/39682/overview/streak",
4   "@representation": "standard",
5   "streak": {
6     "days": 46,
7     "builds": 2
8   }
9 }

```

The Active Record code shown in Listing 5.5 generates this JSON response. The first 9 lines show a subquery which requests the ID of the most recent failed, errored, or canceled build on the default branch. In line 10 this query is surrounded with a COALESCE which replaces a NULL with a 0 in order to allow the selection in line 18 even if no build in this repository has ever resulted in one of the listed statuses.

The next query requests the count (line 11) and date of the first build (line 12) of all passed (line 16) and relevant (line 14 and 15) builds which were created after the last not successful one queried above (line 18). These builds are requested and grouped by the `event_type` in line 19.

Listing 5.5: Active Record code for the streak endpoint in `lib/travis/api/v3/queries/overview.rb`

```

1 def streak(repo)
2   subquery = Models::Build
3     .where(:repository_id => repo.id,
4           :branch => repo.default_branch_name,
5           :state => ['failed', 'canceled', 'errored'],
6           :event_type => ['push', 'cron'])
7     .order("id DESC")
8     .select(:id)
9     .limit(1)
10  subquery = "SELECT COALESCE((" + subquery.to_sql + "), 0)"
11  Models::Build.select('COUNT(*) AS "count",
12                      MIN(created_at) AS "created_at",
13                      "event_type"')
14  .where(:repository_id => repo.id,
15        :branch => repo.default_branch_name,
16        :state => 'passed',
17        :event_type => ['push', 'cron'])
18  .where("id > (#{subquery})")
19  .group(:event_type)
20  .to_a
21 end

```

The `streak_overview` function of the repository model calls this `streak` function (line 2 in Listing 5.6). The loop in line 7 iterates over each grouped `build_type`. To determine the start of the streak the earliest build with success regardless of the type is chosen, see line 8. In the next line the count for the streak is set, but this time only if the type is push.

Build Duration

The JSON with the data for the build duration diagram can be requested from the endpoint `build_history`. Listing 5.7 is an example of a JSON returned. In line 5, one can see that the key to access all data for this endpoint is `build_duration`. This key holds an array with the minimal representations²⁰ of the last 20 builds. This representation includes the number as well as the start and end time.

To return this result we need to get the last 20 builds from the database. This can be accomplished with Active Record as shown in Listing 5.8. Line 2 filters the correct builds. In line 3 are two selections, the first one filters builds without duration which are any builds waiting to be finished. The second filter rejects canceled builds. At the end the first 20 builds are returned descending in their creation time (line 4).

²⁰The Travis API v3 has a minimal and standard representation for returning most of the queryable models. The minimal representation does not always include all attributes and does not expand linked entries. We use the minimal representation because all needed data is included there.

Listing 5.6: Code of the model for the streak endpoint in `lib/travis/api/v3/models/repository.rb`

```

1 def streak_overview
2   result = overview_query.streak(self)
3
4   start_of_streak = DateTime::Infinity.new
5   build_count = 0
6
7   result.each do |builds|
8     start_of_streak = builds.created_at if builds.created_at < start_of_streak
9     build_count = builds.count.to_i if builds.event_type == "push"
10  end
11
12  day_count =
13    (build_count > 0) ? ((Time.now - start_of_streak) / (60*60*24)).floor : 0
14
15  Models::Overview::Streak.new({days: day_count, builds: build_count})
16 end

```

Listing 5.7: JSON returned from the build duration overview endpoint

```

1 {
2   "@type": "overview",
3   "@href": "/v3/repo/39682/overview/build_duration",
4   "@representation": "standard",
5   "build_duration": [
6     {
7       "@type": "build",
8       "@href": "/v3/build/493035",
9       "@representation": "minimal",
10      "id": 493035,
11      "number": "46",
12      "state": "passed",
13      "duration": 164,
14      "event_type": "cron",
15      "previous_state": "passed",
16      "started_at": "2016-04-28T13:46:40Z",
17      "finished_at": "2016-04-28T13:48:01Z"
18    },
19    ...
20  ]
21 }

```

Listing 5.8: Active Record code for the build duration endpoint in `lib/travis/api/v3/queries/overview.rb`

```

1 def build_duration(repo)
2   Models::Build.where(:repository_id => repo.id, :branch => repo.default_branch_name)
3   .where("duration IS NOT NULL").where("state != 'canceled'")
4   .order("id DESC").first(20)
5 end

```

5.5.2 Web

On the overview page the website sends requests to the Travis API to receive the JSON results described in subsection 5.5.1. An example of requesting the data for the `build_duration` is shown in Listing 5.9. The data is requested asynchronously (line 10), while the request is running, a loading indicator symbol is shown (lines 3 and 13). This procedure is used for all statistics we display on the overview page.

The diagrams we display are drawn by `D3.js`,²¹ a JavaScript library for visualization. Since it is arbitrary and not relevant we are not explaining how exactly we use `D3.js`. We chose `D3.js` because it is flexible, runs completely in the clients' browser, and is open source.

Before we are going into the details of each statistic one can see a prototype of the whole new overview page in Figure 5.8. In addition to the statistics described before, we show the branch row of the default branch (3) – the same element used for every branch on the Branches tab. Because the streak just displays two numbers, the number of days passing (4) and the amount of builds during these days (5), we are not explaining any details.

Listing 5.9: Example of loading the data from the Travis API in the ember.js code of the Travis website

```

1 load: function() {
2   // set flag to show loading indicator instead of drawing the chart
3   this.set("isLoading", true);
4
5   // remove old chart, in case of rerendering
6   d3.select("#build_duration_chart").remove();
7
8   ...
9
10  $.ajax(apiEndpoint + "/v3/repo/" + repoId + "/overview/build_duration", options)
11  .then(function(response) {
12    self.set("json", response);
13    self.set("isLoading", false);
14  });
15  return "";
16 }.property("repo"),

```

Build History

The build history chart is shown as first element on the new overview page ((1) in Figure 5.8). When hovering over one bar the site opens a popup which shows the number of builds of each status on this day.

²¹Data-Driven Documents, see <https://d3js.org> (visited on 2016-06-27).

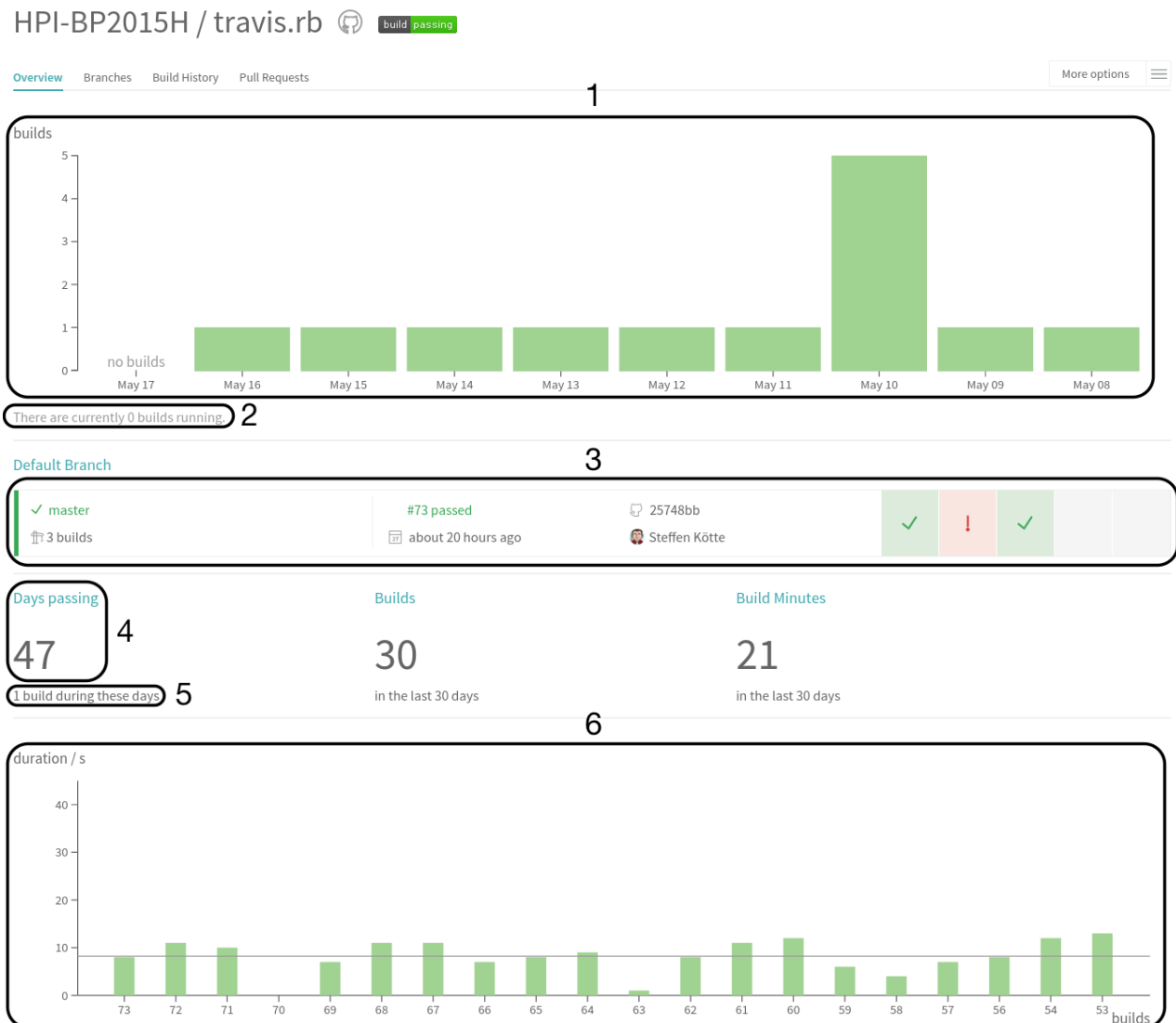


Figure 5.8: Prototype of the new overview page

Beneath the chart is the count of the builds which are queued or running at the moment (2) displayed. If there are any such builds a list with information about these builds as shown in Figure 5.9 is visible below the counter.

Each entry of this list contains an icon to indicate the build type, the number, and state along with a link to the build (1), the already elapsed time (2), branch (3), commit hash with link to GitHub (4), author with Gravatar (5), and the commit message (6). The author combined with the picture offers easy navigation to builds in which the developer is interested in.

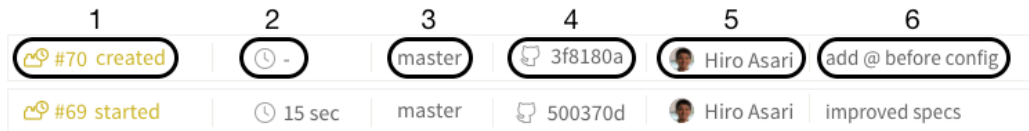


Figure 5.9: Mockup of the list with queued and running builds on the new overview page

Build Duration

The diagram with the build duration is the last element on the overview page ((6) in Figure 5.8). When hovering over one bar a popup with the exact duration of the build is opened. Clicking on one bar opens the site of the corresponding build.

5.6 Evaluation

With our solution introducing the overview page we aimed to solve different issues. First to avoid confusing when the Current tab on the Travis website leads a developer to a build they did not start and second to give developers helpful insights about their project while avoiding the problems of external services.

Avoid Confusion

With removing the Current tab completely, developers cannot be led to an unintended build anymore. By adding the list of currently running builds the functionality is still given and only one click more.

Helpful Insights

Our goal was to offer helpful insights and not to show random statistics. As developers have already built the Buildtime Trend tool, which shows diagrams like our build duration diagram, this diagram has additional value for developers. As the current tab was used before we expect that the list of currently running build is helpful and will be used, too. With all diagrams we tried to stick to the properties of good visualization listed in [38, p. 4].

Avoid Problems of External Services

With integrating our changes directly into the Travis website we wanted to prevent the problems that external services face (see subsection 5.3.2). With the overview tab as start page for a repository developers should neither have a problem finding the statistics nor the need to use different tools. As shown in subsection 5.5.1 the data for our statistics is aggregated in the database so the performance of the overview page is better than that of external services.

Limitations

While discussing this work with several people which use Travis for projects at least on a weekly base, two problems were discovered. First people said that they only use the Travis website if a pull request check is failed to check the reason of the failure. Second, the time axis of the build duration as well as the recent build history diagrams are reversed in comparison with normal time axis.

The axis shows the newest build or date on the left side to be consistent with the already existing elements on the Travis website. In a branch row ((1) in Figure 5.3) the status of the latest build (10) is also displayed on the left side. For developers familiar with the Travis website this might not be a problem but for the one using Travis like mentioned above this is a problem. In order to fix it one needs to change the layout of the branch row. At first it will be confusing to current users, but in the end all diagrams might be easier to understand.

5.7 Related Work

In this section we are presenting how other CI services display their data as well as different ideas on how to show the data generated during development with continuous integration.

5.7.1 CI Tools

Most CI tools already have some sort of data visualization. In the following paragraphs we will introduce some examples of visualization in other CI services.

GitLab CI

GitLab offers an integrated CI service called GitLab CI²² It offers a continuous integration graph page with graphs similar to our build history for the last week, month, and year. On the top of the page the statistics shown in Figure 5.10 are additionally displayed.

²²<https://about.gitlab.com/gitlab-ci/> (visited on 2016-06-28).



Figure 5.10: Screenshot from the continuous integration graph page in GitLab

CircleCI

CircleCI²³ is a CI service similar to Travis. It also displays the build duration and a build history shown in Figure 5.11. It supports JUnit XML to collect metadata from tests [13].

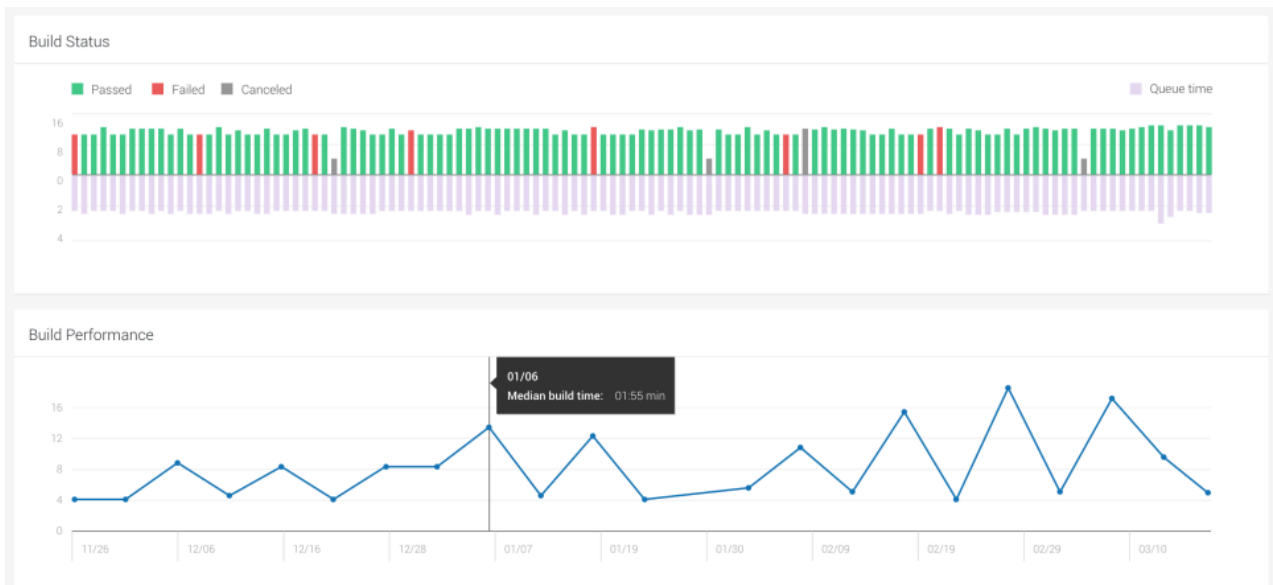


Figure 5.11: CircleCI Insights from [12]

²³<https://circleci.com> (visited on 2016-06-28).

TeamCity

TeamCity²⁴ offers integrated visualization as well as an easy integration to add own charts [42]. It supports data like code duplications, different code coverages, and code inspection errors.

Jenkins

Jenkins²⁵ aggregates the test results and build status of the last builds into a weather report as shown in Figure 5.12.

W	Name ↓	Last Success
	jenkins_2.0	1 mo 11 days - #4
W	Description	%
	Build stability: 2 out of the last 4 builds failed.	50
	Test Result: 0 tests failing out of a total of 11,803 tests.	100 ⁹

Figure 5.12: Weather report from Jenkins Core [27]

5.7.2 Linking different data sources

Data about the software is currently spread across different services like GitHub for issues and CI tools for build statuses. Combining this data can provide a faster overview about the status of a project [8, p. 193] or present information for all stakeholders [34, p. 29]. For example the files changed in commits which break builds might be interesting for further analysis, as developed in [29, pp. 26–28].

5.7.3 Dashboard in Development Rooms

Using physical artifacts like a traffic light to show the last build status in a development room is common [37, p. 4]. Presenting more information than the current build status in a dashboard which is visible the whole time can offer fast insights into the software [8].

5.8 Future Work

The overview page developed in this chapter can be further improved. In the following we are introducing some ideas for enhancements.

²⁴<https://www.jetbrains.com/teamcity/> (visited on 2016-06-29).

²⁵<https://jenkins.io> (visited on 2016-06-29).

JUnit XML

Travis can collect data from individual tests as suggested in subsection 5.4.2. With this information, completely different statistics can improve the benefit of the overview page.

Duration of Pull Request Builds

Before merging a pull request it might be interesting to see how the build time has changed. If this is displayed it can prevent merging code which reduces the performance of the software.

Mobile Website

Our suggested overview page is not yet optimized for mobile devices. If developers use the Travis website often on their smartphones it would be necessary to adjust the graphics to the smaller screens.

5.9 Conclusion

In this chapter we discussed the data collected from tests and the presentation of them on the Travis website. As most data is currently spread over different pages we have suggested an overview page for each repository shown in Figure 5.8.

Our suggestions for the overview page are diagrams with the build duration over time and recent build history, as well as a list of currently running builds and the amount of days without failing builds together with the amount builds during these days. We explain the value to the developer of each statistic and show details of the implementation.

The new overview page enables developers to get the big picture of their repositories on Travis as soon as they open it, for example via the build status badge on GitHub. They can directly navigate to all currently running builds and for example see if the builds of the project have recently gotten slower. With these improvements we make the Travis website more valuable for developers and allow them to get insights which they could not easily get before.

6 Conclusion

In this report we have described the micro-service architecture of Travis CI. Then we have shown how Travis CI can be extended to provide means of coping with dependencies, as well as to provide visualisations of useful information about software projects tested on Travis CI.

With this report we aimed at achieving the following two goals. Firstly, we wanted to help the people developing Travis CI by giving an comprehensible overview of Travis CI's complex architecture. This will facilitate the finding of the correct place for changes when making improvements to Travis CI. Secondly, we wanted to show extensions to Travis CI we found most useful. Therefore we presented periodic building and dependency graphs as means of coping with dependencies, as well as an analyzation of which charts and data are useful for an overview page of a software project. Periodic building has already become a part of Travis CI, while the other two extensions could be implemented with the help of this report.

Since Travis CI is mostly open source, not only staff of Travis CI can benefit from this report. Everyone who wants to make changes to Travis CI can find advice in this work. Therefore this report is a contribution to the entire open source community.

References

- [1] D. Adriaenssens. *Buildtime Trend*. 2015. URL: <https://buildtimetrend.github.io> (visited on 2016-06-08).
- [2] *Android Developer Reference Manual. Scheduling Repeating Alarms*. Android Open Source Project. URL: <https://developer.android.com/training/scheduling/alarms.html> (visited on 2016-06-20).
- [3] *Android Developer Reference Manual. Alarm Manager*. Android Open Source Project. URL: <https://developer.android.com/reference/android/app/AlarmManager.html> (visited on 2016-06-20).
- [4] A. Balalaie, A. Heydarnoori, and P. Jamshidi. "Microservices Architecture Enables DevOps: Migration to a Cloud-Native Architecture". In: *IEEE Software* 33:3 (May 2016), pages 42–52. ISSN: 0740-7459. DOI: 10.1109/MS.2016.64.
- [5] K. Beck and C. Andres. *Extreme Programming Explained: Embrace Change*. Second Edition. Boston: Addison-Wesley, 2004. ISBN: 978-0-321-27865-4.
- [6] S. Bohner and R. Arnold. *Software Change Impact Analysis*. Los Alamitos, CA, USA: IEEE Computer Society Press, 1996. ISBN: 978-0-8186-7384-9.
- [7] G. Booch. *Object Oriented Design: With Applications*. The Benjamin/Cummings Series in Ada and Software Engineering. Benjamin-Cummings Publishing Company, Subs of Addison Wesley Longman, Sept. 1990. ISBN: 978-0-8053-0091-8.
- [8] M. Brandtner, E. Giger, and H. Gall. "Supporting continuous integration by mashing-up software quality information". In: *2014 Software Evolution Week - IEEE Conference on Software Maintenance, Reengineering and Reverse Engineering (CSMR-WCRE)*. 2014, pages 184–193.
- [9] E. Brechner. *Diamond Dependencies*. June 2015. URL: https://blogs.msdn.microsoft.com/eric_brechner/2015/06/30/diamond-dependencies/ (visited on 2016-06-27).
- [10] S.-C. Burcă. *Travis CI Build Statistics*. 2014. URL: <http://scribu.net/blog/travis-ci-build-stats.html> (visited on 2016-06-07).
- [11] Catch Software. *JUnit Format*. 2015. URL: <http://help.catchsoftware.com/display/ET/JUnit+Format> (visited on 2016-05-24).
- [12] Circle CI. *Announcing CircleCI Per-project Insights*. 2016. URL: <https://circleci.com/blog/announcing-circleci-per-project-insights/> (visited on 2016-06-28).
- [13] *Circle CI Documentation. Collecting test metadata*. Circle CI. 2016. URL: <https://circleci.com/docs/test-metadata/> (visited on 2016-06-29).

References

- [14] *Dictionary.com. Bleeding Edge*. Dictionary.com, LLC. 2016. URL: <http://www.dictionary.com/browse/bleeding-edge> (visited on 2016-06-27).
- [15] *Documentation Of The Python Standard Library. Datetime — Basic date and time types*. Python version: 2.7.12. Python Software Foundation. URL: <https://docs.python.org/2/library/datetime.html> (visited on 2016-06-26).
- [16] J. Donald. *Improved Portability of Shared Libraries*. Princeton, NJ 08544, USA, Jan. 2003. URL: http://web.archive.org/web/20070926130800/http://www.princeton.edu/~jdonald/research/shared_libraries/cs518_report.pdf (visited on 2016-06-27).
- [17] P. Edberg et al. *Unicode Locale Data Markup Language*. Unicode Technical Standard 35. version: 29. The Unicode Consortium, Mar. 2016. URL: <http://www.unicode.org/reports/tr35/tr35-43/tr35-dates.html> (visited on 2016-06-24).
- [18] C. Escoffier. *Building pipelines by linking Jenkins jobs*. Nov. 2011. URL: <https://blog.akquinet.de/2011/11/09/building-pipelines-by-linking-jenkins-jobs/> (visited on 2016-06-27).
- [19] M. Fowler. *Continuous Integration*. May 2006. URL: <http://www.martinfowler.com/articles/continuousIntegration.html> (visited on 2016-06-27).
- [20] M. Fowler. *Recurring Events for Calendars*. May 1996. URL: <http://martinfowler.com/apsupp/recurring.pdf> (visited on 2016-06-24).
- [21] S. Fuchs. *Travis - an experimental, distributed CI server on Heroku*. June 2010. URL: <http://svenfuchs.com/2010/6/16/travis-an-experimental-distributed-ci-server-on-heroku> (visited on 2016-06-24).
- [22] *GitHub Press Resources*. GitHub Inc. 2016. URL: <https://github.com/about/press> (visited on 2016-06-27).
- [23] *GoCD User Documentation. Managing dependencies*. ThoughtWorks, Inc. 2016. URL: https://docs.go.cd/current/configuration/managing_dependencies.html (visited on 2016-06-27).
- [24] E. Goldin. *TeamCity Build Dependencies*. JetBrains. Apr. 2012. URL: <https://blog.jetbrains.com/teamcity/2012/04/teamcity-build-dependencies-2/> (visited on 2016-06-27).
- [25] D. R. Hofstadter. *Gödel, Escher, Bach: An Eternal Golden Braid*. New York, NY, USA: Basic Books, Inc., 1979. ISBN: 978-0-465-02685-2.
- [26] T. Howard. *JUnit-Schema*. 2015. URL: <https://github.com/windyroad/JUnit-Schema/blob/master/JUnit.xsd> (visited on 2016-05-24).
- [27] *Jenkins on Jenkins*. Jenkins. 2016. URL: <https://ci.jenkins.io/job/Core/> (visited on 2016-06-29).
- [28] *Jenkins Wiki. Overview of the Build Pipeline Plugin for Jenkins*. Jenkins. URL: <https://wiki.jenkins-ci.org/display/JENKINS/Build+Pipeline+Plugin> (visited on 2016-06-27).

- [29] S. Judis. "Visualisierung und Analyse von Automated-Build-Data am Beispiel von Travis CI". German. Bachelor's Thesis. Hochschule für Technik und Wirtschaft Berlin, 2013.
- [30] G. Klyne and C. Newman. *Date and Time on the Internet: Timestamps*. RFC 3339. Internet Request for Comments. Internet Engineering Task Force, July 2002. URL: <https://www.ietf.org/rfc/rfc3339.txt> (visited on 2016-06-24).
- [31] M. Lipper. *Ruby Temporal Expressions*. URL: <https://github.com/mlipper/run/blob/7ab3887581fa962ba04890fb21bc33871bc48240/README.md> (visited on 2016-06-21).
- [32] M. Lipper. *Temporal Expressions Tutorial*. URL: https://github.com/mlipper/run/blob/7ab3887581fa962ba04890fb21bc33871bc48240/doc/tutorial_te.md (visited on 2016-06-21).
- [33] R. C. Martin. "Java and C++: A Critical Comparison". In: *Java Gems*. Edited by D. Deugo. New York, NY, USA: Cambridge University Press, 1998, pages 51–68. ISBN: 978-0-521-64824-0.
- [34] A.-L. Mattila, T. Lehtonen, H. Terho, T. Mikkonen, and K. Systä. "Mashing Up Software Issue Management, Development, and Usage Data". In: *Proceedings of the Second International Workshop on Rapid Continuous Software Engineering*. RCoSE '15. Florence, Italy: IEEE Press, 2015, pages 26–29.
- [35] M. Meyer. *An Update on Infrastructure Changes*. Dec. 2012. URL: <https://blog.travis-ci.com/2012-12-13-an-update-on-infrastructure-changes> (visited on 2016-06-24).
- [36] *Oxford Dictionaries*. *Dependence*. Oxford University Press, 2016. URL: <http://www.oxforddictionaries.com/definition/english/dependence> (visited on 2016-06-27).
- [37] J. Paredes, C. Anslow, and F. Maurer. "Information Visualization for Agile Software Development". In: *2014 Second IEEE Working Conference on Software Visualization (VISSOFT)*. Sept. 2014, pages 157–166. DOI: 10.1109/VISSOFT.2014.32.
- [38] M. Petre, E. de Quincey, et al. "A gentle overview of software visualisation". In: *PPIG News Letter* (2006), pages 1–10.
- [39] *Ruby Documentation*. *Date*. Ruby version: 1.9.3. Ruby Documentation Project. URL: <http://ruby-doc.org/stdlib-1.9.3/libdoc/date/rdoc/Date.html> (visited on 2016-06-26).
- [40] H. Schumann and W. Müller. *Visualisierung: Grundlagen und allgemeine Methoden*. German. Springer Berlin Heidelberg, 1999. ISBN: 978-3-540-64944-1.
- [41] D. Shakib, S. Sundararaman, D. Cornfield, S. Alam, and D. Whitney. *Representing recurring events*. US Patent 5,813,013, Microsoft Corporation, Redmond, Wash. Sept. 1998.

References

- [42] *TeamCity Documentation. Custom Chart*. JetBrains. 2016. URL: <https://confluence.jetbrains.com/display/TCD9/Custom+Chart> (visited on 2016-06-29).
- [43] J. Tellnes. "Dependencies: No Software is an Island". Master's thesis. University of Bergen, 2013.
- [44] *The Open Group Base Specifications Issue 7, IEEE Std 1003.1. Crontab*. The IEEE and The Open Group. 2013. URL: <http://pubs.opengroup.org/onlinepubs/9699919799/utilities/crontab.html> (visited on 2016-06-24).
- [45] J. Thompson. *Dependency Hell*. May 2005. URL: https://www.haiku-os.org/documents/dev/dependency_hell (visited on 2016-06-27).
- [46] *Travis Documentation. Customizing the Build*. Travis CI GmbH. 2016. URL: <https://docs.travis-ci.com/user/customizing-the-build/> (visited on 2016-06-24).
- [47] *Travis Documentation. Languages*. Travis CI GmbH. 2016. URL: <https://docs.travis-ci.com/user/languages/> (visited on 2016-07-05).
- [48] P. Vixie. *FreeBSD File Formats Manual. Cron – daemon to execute scheduled commands (Vixie Cron)*. The FreeBSD Project. June 2008. URL: [https://www.freebsd.org/cgi/man.cgi?cron\(8\)](https://www.freebsd.org/cgi/man.cgi?cron(8)) (visited on 2016-06-24).
- [49] P. Vixie. *FreeBSD File Formats Manual. Crontab – tables for driving cron*. The FreeBSD Project. Apr. 2012. URL: [https://www.freebsd.org/cgi/man.cgi?crontab\(5\)](https://www.freebsd.org/cgi/man.cgi?crontab(5)) (visited on 2016-06-24).

Aktuelle Technische Berichte des Hasso-Plattner-Instituts

Band	ISBN	Titel	Autoren / Redaktion
107	978-3-86956-373-2	Extending a dynamic programming language and runtime environment with access control	Philipp Tessenow, Tim Felgentreff, Gilad Bracha, Robert Hirschfeld
106	978-3-86956-372-5	On the Operationalization of Graph Queries with Generalized Discrimination Networks	Thomas Beyhl, Dominique Blouin, Holger Giese, Leen Lambers
105	978-3-86956-360-2	Proceedings of the Third HPI Cloud Symposium "Operating the Cloud" 2015	Estee van der Walt, Jan Lindemann, Max Plauth, David Bartok (Hrsg.)
104	978-3-86956-355-8	Tracing Algorithmic Primitives in RSqueak/VM	Lars Wassermann, Tim Felgentreff, Tobias Pape, Carl Friedrich Bolz, Robert Hirschfeld
103	978-3-86956-348-0	Babelsberg/RML : executable semantics and language testing with RML	Tim Felgentreff, Robert Hirschfeld, Todd Millstein, Alan Borning
102	978-3-86956-347-3	Proceedings of the Master Seminar on Event Processing Systems for Business Process Management Systems	Anne Baumgraß, Andreas Meyer, Mathias Weske (Hrsg.)
101	978-3-86956-346-6	Exploratory Authoring of Interactive Content in a Live Environment	Philipp Otto, Jaqueline Pollak, Daniel Werner, Felix Wolff, Bastian Steinert, Lauritz Thamsen, Macel Taeumel, Jens Lincke, Robert Krahn, Daniel H. H. Ingalls, Robert Hirschfeld
100	978-3-86956-345-9	Proceedings of the 9th Ph.D. retreat of the HPI Research School on service-oriented systems engineering	Christoph Meinel, Hasso Plattner, Jürgen Döllner, Mathias Weske, Andreas Polze, Robert Hirschfeld, Felix Naumann, Holger Giese, Patrick Baudisch, Tobias Friedrich (Hrsg.)
99	978-3-86956-339-8	Efficient and scalable graph view maintenance for deductive graph databases based on generalized discrimination networks	Thomas Beyhl, Holger Giese
98	978-3-86956-333-6	Inductive invariant checking with partial negative application conditions	Johannes Dyck, Holger Giese

ISBN 978-3-86956-377-0
ISSN 1613-5652