tsuru Documentation

Release 0.12.3

timeredbull

October 07, 2015

Contents

tsuru is an open source PaaS that makes it easy and fast to deploy and manage applications on your own servers.

To get started, first read *understanding tsuru*.

CHAPTER 1

Understanding

1.1 Overview

tsuru is an extensible and open source Platform as a Service (PaaS) that makes application deployments faster and easier. tsuru is an open source polyglot cloud application platform (PaaS). With tsuru, you don't need to think about servers at all. As an application developer, you can:

- Write apps in the programming language of your choice,
- Back apps with add-on resources such as SQL and NoSQL databases, including memcached, redis, and many others.
- Manage apps using the `tsuru` command-line tool
- Deploy apps using the Git revision control system

1.1.1 Why tsuru?

Fast and easy and continuous deployment

Deploying an app is simple and easy. No special tools needed, just a plain git push. The entire process is very simple. tsuru will also take care of all the applications dependencies in the deployment process.

Easily create testing, staging, and production versions of your app and deploy to them instantly.

Scaling

Scaling applications is completely painless. Just add a unit and tsuru will take care of everything else.

Reliable

tsuru has a set of tools to make sure that the applications will be always available.

Open source

tsuru is free, open source software released under the BSD 3-Clause license.

1.2 Concepts

1.2.1 Docker

Docker is an open source project to pack, ship, and run any application as a lightweight, portable, self-sufficient container. When you deploy an app with `git push` or `tsuru app-deploy`, tsuru builds a Docker image and then distributes it as *units* (Docker containers) across your cluster.

1.2.2 Clusters

A cluster is a named group of nodes. *tsuru API* has a scheduler algorithm that distributes applications intelligently across a cluster of nodes.

1.2.3 Nodes

A node is a physical or virtual machine with Docker installed.

Managed node

A managed node is a node created and managed by tsuru, using *IaaS integration*. tsuru manages this node, i.e. tsuru can heal and scale it.

Unmanaged node

An unmanaged node is a node created manually, and just registered with tsuru. tsuru is not able to manage these nodes, and it should be handled by whoever created it manually.

1.2.4 Applications

An application consists of:

- the program's source code - e.g.: Python, Ruby, Go, PHP, JavaScript, Java, etc.
- an operating system dependencies list – in a file called `requirements.apt`
- a language-level dependencies list – e.g.: `requirements.txt`, `Gemfile`, etc.
- instructions on how to run the program – in a file called `Procfile`

An application has a name, a unique address, a platform, associated development teams, a repository, and a set of units.

1.2.5 Units

A unit is a container. A unit has everything an application needs to run; the fetched operational system and language level dependencies, the application's source code, the language runtime, and the application's processes defined in the `Procfile`.

1.2.6 Platforms

A platform is a well-defined pack with installed dependencies for a language or framework that a group of applications will need. A platform might be a container template (Docker image).

For instance, tsuru has a container image for Python applications, with virtualenv installed and other required things needed for tsuru to deploy applications on top of that platform. Platforms are easily extendable and managed by tsuru. Every application runs on top of a platform.

1.2.7 Services

A service is a well-defined API that tsuru communicates with to provide extra functionality for applications. Examples of services are MySQL, Redis, MongoDB, etc. tsuru has built-in services, but it is easy to create and add new services to tsuru. Services aren't managed by tsuru, but by their creators.

1.3 Architecture

1.3.1 API

The API component (also referred as the tsuru daemon, or *tsurud*) is a RESTful API server written with `Go`. The API is responsible for the deploy workflow and the lifecycle of applications.

Command-line clients interact with this component.

1.3.2 Database

The database component is a *MongoDB* server.

1.3.3 Queue/Cache

The queue and cache component uses *Redis*.

1.3.4 Gandalf

Gandalf is a REST API to manage Git repositories and users and provides access to them over SSH.

1.3.5 Registry

The registry component hosts *Docker* images.

1.3.6 Router

The router component routes traffic to application units (Docker containers).

CHAPTER 2

Installing

If you want to try tsuru with a minimum amount of effort, we recommend you to use tsuru Now (or tsuru-bootstrap, which runs tsuru Now in a Vagrant VM).

tsuru Now will install tsuru API, tsuru Client, tsuru Admin, and all of their dependencies on a single machine. It will also include a Docker node which will run deployed applications.

This gives you a very nice environment for trying out tsuru, but this is not the recommended approach for a production environment. This document will describe how to install each component separately.

We assume that tsuru is being installed on an Ubuntu Server 14.04 LTS 64-bit machine. This is currently the supported environment for tsuru, you may try running it on other environments, but there's a chance it won't be a smooth ride.

2.1 Gandalf

tsuru optionally uses Gandalf to manage Git repositories used to push applications to. It's also responsible for setting hooks in these repositories which will notify the tsuru API when a new deploy is made. For more details check Gandalf Documentation

This document will focus on how to setup a Gandalf installation with the necessary hooks to notify the tsuru API.

2.1.1 Adding repositories

Let's start adding the repositories for tsuru which contain the Gandalf package.

```
sudo apt-get update
sudo apt-get install curl python-software-properties
sudo apt-add-repository ppa:tsuru/ppa -y
sudo apt-get update
```

2.1.2 Installing

```
sudo apt-get install gandalf-server
```

A deploy is executed in the `git push`. In order to get it working, you will need to add a pre-receive hook. tsuru comes with three pre-receive hooks, all of them need further configuration:

- s3cmd: uses Amazon S3 to store and serve archives
- archive-server: uses tsuru's archive-server to store and serve archives

- swift: uses Swift to store and serve archives (compatible with Rackspace Cloud Files)

In this documentation, we will use archive-server, but you can use anything that can store a git archive and serve it via HTTP or FTP. You can install archive-server via apt-get too:

```
sudo apt-get install archive-server
```

Then you will need to configure Gandalf, install the pre-receive hook, set the proper environment variables and start Gandalf and the archive-server, please note that you should replace the value `<your-machine-addr>` with your machine public address:

```
sudo mkdir -p /home/git/bare-template/hooks
sudo curl https://raw.githubusercontent.com/tsuru/tsuru/master/misc/git-hooks/pre-receive.archive-ser
sudo chmod +x /home/git/bare-template/hooks/pre-receive
sudo chown -R git:git /home/git/bare-template
cat | sudo tee -a /home/git/.bash_profile <<EOF
export ARCHIVE_SERVER_READ=http://<your-machine-addr>:3232 ARCHIVE_SERVER_WRITE=http://127.0.0.1:3131
EOF
```

In the `/etc/gandalf.conf` file, remove the comment from the line "template: /home/git/bare-template", so it looks like that:

```
git:
  bare:
    location: /var/lib/gandalf/repositories
    template: /home/git/bare-template
```

Then start gandalf and archive-server:

```
sudo start gandalf-server
sudo start archive-server
```

2.1.3 Configuring tsuru to use Gandalf

In order to use Gandalf, you need to change tsuru.conf accordingly:

1. Define "repo-manager" to use "gandalf";
2. Define "git:api-server" to point to the API of the Gandalf server (example: "http://localhost:8000");

For more details, please refer to the configuration page.

2.1.4 Token for authentication with tsuru API

There is one last step in configuring Gandalf. It involves generating an access token so that the hook we created can access the tsuru API. This must be done after installing the tsuru API and it's detailed in the next *installation step*.

2.2 API Server

2.2.1 Dependencies

tsuru API depends on a MongoDB server, Redis server, Hipache router, and Gandalf server. Instructions for installing MongoDB and Redis are outside the scope of this documentation, but it's pretty straight-forward following their docs. Installing Gandalf and installing Hipache are described in other sessions.

2.2.2 Adding repositories

Let's start adding the repositories for tsuru.

```
sudo apt-get update
sudo apt-get install python-software-properties
sudo apt-add-repository ppa:tsuru/ppa -y
sudo apt-get update
```

2.2.3 Installing

```
sudo apt-get install tsuru-server -qqy
```

Now you need to customize the configuration in the `/etc/tsuru/tsuru.conf`. A description of possible configuration values can be found in the configuration reference. A basic possible configuration is described below, please note that you should replace the values `your-mongodb-server`, `your-redis-server`, `your-gandalf-server` and `your-hipache-server`.

```
listen: "0.0.0.0:8080"
debug: true
host: http://<machine-public-addr>:8080 # This port must be the same as in the "listen" conf
admin-team: admin
auth:
    user-registration: true
    scheme: native
database:
    url: <your-mongodb-server>:27017
    name: tsurudb
pubsub:
    redis-host: <your-redis-server>
    redis-port: 6379
queue:
    mongo-url: <your-mongodb-server>:27017
    mongo-database: queuedb
git:
    api-server: http://<your-gandalf-server>:8000
provisioner: docker
docker:
    router: hipache
    collection: docker_containers
    repository-namespace: tsuru
    deploy-cmd: /var/lib/tsuru/deploy
    cluster:
        storage: mongodb
        mongo-url: <your-mongodb-server>:27017
        mongo-database: cluster
    run-cmd:
        bin: /var/lib/tsuru/start
        port: "8888"
    ssh:
        add-key-cmd: /var/lib/tsuru/add-key
        user: ubuntu
routers:
    hipache:
        type: hipache
        domain: <your-hipache-server-ip>.xip.io
        redis-server: <your-redis-server-with-port>
```

In particular, take note that you must set `auth:user-registration` to `true`:

```
auth:
    user-registration: true
    scheme: native
```

Otherwise, tsuru will fail to create an admin user in the next section.

Now you only need to start your tsuru API server:

```
sudo sed -i -e 's/=no/=yes/' /etc/default/tsuru-server
sudo start tsuru-server-api
```

2.2.4 Creating admin user

The creation of an admin user is necessary for the next steps, so we're going to describe how to install the `tsuru` and create a new user belonging to the admin team configured in your `tsuru.conf` file. For a description of each command shown below please refer to the client documentation.

For a description

```
$ sudo apt-get install tsuru-client

$ tsuru target-add default http://<your-tsuru-api-addr>:8080
$ tsuru target-set default
$ tsuru user-create myemail@somewhere.com
# type a password and confirmation

$ tsuru login myemail@somewhere.com
# type the chosen password

$ tsuru team-create admin
```

And that's it, you now have registered a user in your tsuru API server ready to run admin commands.

2.2.5 Generating token for Gandalf authentication

Assuming you have already configured your Gandalf server in the previous installation step, now we need to export two extra environment variables to the git user, which will run our deploy hooks, the URL to our API server and a generated token.

First step is to generate a token in the machine we've just installed the API server:

```
$ tsurud token
fed1000d6c05019f6550b20dbc3c572996e2c044
```

Now you have to go back to the machine you installed Gandalf, and run this:

```
$ cat | sudo tee -a /home/git/.bash_profile <<EOF
export TSURU_HOST=http://<your-tsuru-api-addr>:8080
export TSURU_TOKEN=fed1000d6c05019f6550b20dbc3c572996e2c044
EOF
```

2.3 Hipache Router

Hipache is a distributed HTTP and websocket proxy.

tsuru uses Hipache to route the requests to the containers. Routing information is stored by tsuru in the configured Redis server, Hipache will read this configuration directly from Redis.

2.3.1 Adding repositories

Let's start adding the repositories for tsuru which contain the Hipache package.

```
sudo apt-get update
sudo apt-get install python-software-properties
sudo apt-add-repository ppa:tsuru/ppa -y
sudo apt-get update
```

2.3.2 Installing

In order to install Hipache, just use apt-get:

```
sudo apt-get install node-hipache
```

2.3.3 Configuring

In your `/etc/hipache.conf` file you must set the `redisHost` and `redisPort` configuration values. After this, you only need to start Hipache with:

```
sudo start hipache
```

2.4 Adding Nodes

Nodes are physical or virtual machines with a Docker installation.

Nodes can be either unmanaged, which mean that they were created manually, by provisioning a machine and installing Docker on it, in which case they have to be registered in tsuru. Or they can be automatically managed by tsuru, which will handle machine provisioning and Docker installation using your *IaaS configuration*.

The managed option is preferred starting with tsuru-server 0.6.0. There are advantages like automatically healing and scaling of Nodes. The sections below describe how to add managed and unmanaged nodes.

2.4.1 Managed nodes

First step is configuring your IaaS provider in your tsuru.conf file. Please see the details in *IaaS configuration*

Assuming you're using EC2, the configuration will be something like:

```
iaas:
  default: ec2
  node-protocol: http
  node-port: 2375
  ec2:
    key-id: xxxxxxxxxxx
    secret-key: yyyyyyyyyyyyy
```

After you have everything configured, adding a new Docker node is done by calling docker-node-add in tsuru-admin command. This command will receive a map of key=value params which are IaaS dependent. A list of possible key params can be seen calling:

```
$ tsuru-admin docker-node-add iaas=ec2

EC2 IaaS required params:
  image=<image id>          Image AMI ID
  type=<instance type>      Your template uuid

Optional params:
  region=<region>           Chosen region, defaults to us-east-1
  securityGroup=<group>     Chosen security group
  keyName=<key name>        Key name for machine
```

Every key=value pair will be added as a metadata to the Node and you can send After registering your node, you can list it calling tsuru-admin docker-node-list

```
$ tsuru-admin docker-node-add iaas=ec2 image=ami-dc5387b4 region=us-east-1 type=m1.small securityGrou
Node successfully registered.
$ tsuru-admin docker-node-list
+-----------------------------------------------------------+------------+---------+-------------------
| Address                                                   | IaaS ID    | Status  | Metadata
+-----------------------------------------------------------+------------+---------+-------------------
| http://ec2-xxxxxxxxxxxx.compute-1.amazonaws.com:2375 | i-xxxxxxxx | waiting | iaas=ec2
|                                                           |            |         | image=ami-dc5387b4
|                                                           |            |         | keyName=my-key
|                                                           |            |         | region=us-east-1
|                                                           |            |         | securityGroup=my-sec
|                                                           |            |         | type=m1.small
+-----------------------------------------------------------+------------+---------+-------------------
```

2.4.2 Unmanaged nodes

To add a previously provisioned node you call the tsuru-admin docker-node-add with the `--register` flag and setting the address key with the URL of the Docker API in the remote node.

The docker API must be responding in the referenced address. To instructions about how to install docker on your node, please refer to Docker documentation.

```
$ tsuru-admin docker-node-add --register address=http://node.address.com:2375
```

CHAPTER 3

Managing

3.1 Installing platforms

A platform is a well defined pack with installed dependencies for a language or framework that a group of applications will need.

Platforms are defined as Dockerfiles and tsuru already have a number of supported ones listed in https://github.com/tsuru/basebuilder

These platforms don't come pre-installed in tsuru, you have to add them to your server using the platform-add command in tsuru-admin.

```
tsuru-admin platform-add platform-name --dockerfile dockerfile-url
```

For example, to install the Python platform from tsuru's basebuilder repository you simply have to call:

```
tsuru-admin platform-add python --dockerfile https://raw.githubusercontent.com/tsuru/basebuilder/mast
```

Attention: If you have more than one Docker node, you may use docker-registry to add and distribute your platforms among your Docker nodes.
You can use the official docker registry or install it by yourself. To do this you should first have to install docker-registry in any server you have. It should have a public ip to communicate with your docker nodes.
Then you should add registry address to tsuru.conf.

3.2 Creating a platform

3.2.1 Overview

If you need a platform that's not already available in our platforms repository it's pretty easy to create a new one based on a existing one.

To tsuru to be able to use your platform you only need to have the following scripts available on **/var/lib/tsuru**:

- /var/lib/tsuru/deploy
- /var/lib/tsuru/start

3.2.2 Using Docker

Now we will create a whole new platform with Docker, circus and tsuru basebuilder. tsuru basebuilder provides to us some useful scripts like **install, setup and start**.

So, using the base platform provided by tsuru we can write a Dockerfile like that:

```
from ubuntu:14.04
run  apt-get install wget -y --force-yes
run  wget http://github.com/tsuru/basebuilder/tarball/master -O basebuilder.tar.gz --no-check-certif
run  mkdir /var/lib/tsuru
run  tar -xvf basebuilder.tar.gz -C /var/lib/tsuru --strip 1
run  cp /var/lib/tsuru/base/start /var/lib/tsuru
run  cp /home/your-user/deploy /var/lib/tsuru
run  /var/lib/tsuru/base/install
run  /var/lib/tsuru/base/setup
```

3.2.3 Adding your platform to tsuru

After creating you platform as a Docker image, you can add it to tsuru using `tsuru-admin`:

```
$ tsuru-admin platform-add your-platform-name --dockerfile http://url-to-dockerfile
```

3.3 Using Pools

3.3.1 Overview

Pool is used by provisioners to group nodes and know if an application can be deployed in these nodes. Users can choose which pool to deploy in *tsuru app-create*.

Tsuru has three types of pool: team, public and default.

Team's pool are segregated by teams, and cloud administrator should set teams in this pool manually. This pool are just accessible by team's members.

Public pools are accessible by any user.

Default pool is where apps are deployed when app's team owner don't have a pool associated with it or when app's creator don't choose any public pool. Ideally this pool is for experimentation and low profile apps, like service dashboard and "in development" apps. You can just have one default pool. This is the old fallback pool, but with a explicit flag.

Adding a pool

In order to create a pool, you should invoke *tsuru-admin pool-add*:

```
$ tsuru-admin pool-add pool1
```

If you want to create a public pool you can do:

```
$ tsuru-admin pool-add pool1 -p
```

If you want a default pool, you can create it with:

```
$ tsuru-admin pool-add pool1 -d
```

You can overwrite default pool by setting the flag *-f*:

```
$ tsuru-admin pool-add new-default-pool -d -f
```

Adding teams to a pool

Then you can use *tsuru-admin pool-teams-add* to add teams to the pool that you've just created:

```
$ tsuru-admin pool-teams-add pool1 team1 team2

$ tsuru-admin pool-teams-add pool2 team3
```

Listing pools

To list pools you do:

```
$ tsuru-admin pool-list
+-------+-------------+
| Pools | Teams       |
+-------+-------------+
| pool1 | team1 team2 |
| pool2 | team3       |
+-------+-------------+
```

Removing a pool

If you want to remove a pool, use *tsuru-admin pool-remove*:

```
$ tsuru-admin pool-remove pool1
```

Removing teams from a pool

You can remove one or more teams from a pool using the command *tsuru-admin pool-teams-remove*:

```
$ tsuru-admin pool-teams-remove pool1 team1

$ tsuru-admin pool-teams-remove pool1 team1 team2 team3
```

3.4 Segregate Scheduler

3.4.1 Overview

tsuru uses schedulers to chooses which node an unit should be deployed. Previously there was a choice between *round robin* and *segregate scheduler*. As of 0.11.1, only *segregate scheduler* is available and it's the default choice. This change was made because *round robin* scheduler was broken, unmaintained and was a worse scheduling mechanism than *segregate scheduler*.

3.4.2 How it works

Segregate scheduler is a scheduler that segregates the units among pools.

First, what you need to do is to define a relation between a pool and teams. After that you need to register nodes with the `pool` metadata information, indicating to which pool the node belongs.

When deploying an application, the scheduler will choose among the nodes within the application pool.

Registering a node with pool metadata

You can use the `tsuru-admin` with `docker-node-add` to register or create nodes with the pool metadata:

```
$ tsuru-admin docker-node-add --register address=http://localhost:2375 pool=pool1
```

3.5 Upgrading Docker

A *node* is a physical or virtual machine with Docker installed. The nodes should contains one or more units (containers).

Sometimes will be necessary to upgrade the Docker. It is recommended that you use the latest Docker version.

The simple way to do it is just upgrade Docker. You can do it following the official guide.

This operation can cause some period of downtime in an application.

3.5.1 How to upgrade Docker with no application downtime

Note: You should use this guide to upgrade the entire host (a new version of the Linux distro, for instance) or Docker itself.

A way to upgrade with no downtime is to move all containers from the node that you want to upgrade to another node, upgrade the node and then move the containers back.

You can do it using the command tsuru-admin containers-move:

```
$ tsuru-admin containers-move <from host> <to host>
```

3.6 Managing Git repositories and SSH keys

There are two deployment flavors in tsuru: using `git push` and `tsuru app-deploy`. The former is optional, while the latter will always be available. This document focus on the usage of the Git deployment flavor.

In order to allow tsuru users to use `git push` for deployments, tsuru administrators need to install and configure Gandalf.

Gandalf will store and manage all Git repositories and SSH keys, as well as users. When tsuru is configured to use Gandalf, it will interact with the Gandalf API in the following actions:

- When creating a new user in tsuru, a corresponding user will be created in Gandalf;
- When removing a user from tsuru, the corresponding user will be removed from Gandalf;

- When creating an app in tsuru, a new repository for the app will be created in Gandalf. All users in the team that owns the app will be authorized to access this repository;
- When removing an app, the corresponding repository will be removed from Gandalf;
- When adding a user to a team in tsuru, the corresponding user in Gandalf will gain access to all repositories matching the applications that the team has access to;
- When removing a user from a team in tsuru, the corresponding user in Gandalf will lose access to the repositories that he/she has access to because of the team he/she is leaving;
- When adding a team to an application in tsuru, all users from the team will gain access to the repository matching the app;
- When removing a team from an application in tsuru, all users from the team will lose access to the repository, unless they're in another team that also have access to the application.

When user runs a `git push`, the communication happens directly between the user host and the Gandalf host, and Gandalf will notify tsuru the new deployment using a git hook.

3.6.1 Managing SSH public keys

In order to be able to send git pushes to the Git server, users need to have their key registered in Gandalf. When Gandalf is enabled, tsuru will enable the usage of three commands for SSH public keys management:

- tsuru key-add
- tsuru key-remove
- tsuru key-list

Each of these commands have a corresponding API endpoint, so other clients of tsuru can also manage keys through the API.

tsuru will not store any public key data, all the data related to SSH keys is handled by Gandalf alone, and when Gandalf is not enabled, those key commands will not work.

3.6.2 Adding Gandalf to an already existing tsuru cluster

In the case of an old tsuru cluster running without Gandalf, users and applications registered in tsuru won't be available in the newly created Gandalf server, or both servers may be out-of-sync.

When Gandalf is enabled, administrators of the cloud can run the `tsr gandalf-sync` command.

CHAPTER 4

Using

4.1 Installing tsuru clients

tsuru contains three clients: `tsuru`, `tsuru-admin` and `crane`.

- **tsuru** is the command line utility used by application developers, that will allow users to create, list, bind and manage apps. For more details, check tsuru usage;
- **crane** is used by service administrators.
- **tsuru-admin** is used by cloud administrators. Whoever is allowed to use it has gotten super powers :-)

This document describes how you can install those clients, using pre-compiled binaries, packages or building them from source.

4.1.1 Downloading binaries (Mac OS X, Linux and Windows)

We provide pre-built binaries for OS X and Linux, only for the amd64 architecture. You can download these binaries directly from the releases page of the project:

- crane: https://github.com/tsuru/crane/releases
- tsuru: https://github.com/tsuru/tsuru-client/releases
- tsuru-admin: https://github.com/tsuru/tsuru-admin/releases

4.1.2 Using homebrew (Mac OS X only)

If you use Mac OS X and homebrew, you may use a custom tap to install `tsuru`, `crane` and `tsuru-admin`. First you need to add the tap:

```
$ brew tap tsuru/homebrew-tsuru
```

Now you can install tsuru, tsuru-admin and crane:

```
$ brew install tsuru
$ brew install tsuru-admin
$ brew install crane
```

Whenever a new version of any of tsuru's clients is out, you can just run:

```
$ brew update
$ brew upgrade <formula> # tsuru/tsuru-admin/crane
```

For more details on taps, check homebrew documentation.

NOTE: tsuru clients require Go 1.4 or higher. Make sure you have the last version of Go installed in your system.

4.1.3 Using the PPA (Ubuntu only)

Ubuntu users can install tsuru clients using `apt-get` and the tsuru PPA. You'll need to add the PPA repository locally and run an `apt-get update`:

```
$ sudo apt-add-repository ppa:tsuru/ppa
$ sudo apt-get update
```

Now you can install tsuru's clients:

```
$ sudo apt-get install tsuru-client
$ sudo apt-get install crane
$ sudo apt-get install tsuru-admin
```

4.1.4 Build from source (Linux, Mac OS X and Windows)

Note: If you're feeling adventurous, you can try it on other platforms, like FreeBSD and OpenBSD. Please let us know about your progress!

tsuru's source is written in Go, so before installing tsuru from source, please make sure you have installed and configured Go.

With Go installed and configured, you can use `go get` to install any of tsuru's clients:

```
$ go get github.com/tsuru/tsuru-client/tsuru
$ go get github.com/tsuru/tsuru-admin
$ go get github.com/tsuru/crane
```

4.2 Building your app in tsuru

tsuru is an open source polyglot cloud application platform. With tsuru, you don't need to think about servers at all. You:

- Write apps in the programming language of your choice
- Back it with add-on resources (tsuru calls these *services*) such as SQL and NoSQL databases, memcached, redis, and many others.
- Manage your app using the `tsuru` command-line tool
- Deploy code using the Git revision control system

tsuru takes care of where in your cluster to run your apps and the services they use. You can then focus on making your apps awesome.

4.2.1 Install the tsuru client

Install the tsuru client for your development platform.

The `tsuru` client is a command-line tool for creating and managing apps. Check out the CLI usage guide to learn more.

4.2.2 Sign up

To create an account, you use the command *user-create*:

```
$ tsuru user-create youremail@domain.com
```

`user-create` will ask for the desired password twice.

4.2.3 Login

To login in tsuru, you use the command *login*:

```
$ tsuru login youremail@domain.com
```

It will ask for your password. Unless your tsuru installation is configured to use OAuth.

4.2.4 Deploy an application

Choose from the following getting started tutorials to learn how to deploy your first application using one of the supported platforms:

- Deploying Python applications in tsuru
- Deploying Ruby/Rails applications in tsuru
- Deploying PHP applications in tsuru
- Deploying go applications in tsuru

4.3 Deploying Python applications in tsuru

4.3.1 Overview

This document is a hands-on guide to deploying a simple Python application in tsuru. The example application will be a very simple Django project associated to a MySQL service. It's applicable to any WSGI application.

4.3.2 Creating the app within tsuru

To create an app, you use the command *app-create*:

```
$ tsuru app-create <app-name> <app-platform>
```

For Python, the app platform is, guess what, `python`! Let's be over creative and develop a never-developed tutorial-app: a blog, and its name will also be very creative, let's call it "blog":

```
$ tsuru app-create blog python
```

To list all available platforms, use the command *platform-list*.

You can see all your applications using the command *app-list*:

```
$ tsuru app-list
+-------------+-------------------------+--------------------------+
| Application | Units State Summary     | Address                  |
+-------------+-------------------------+--------------------------+
| blog        | 0 of 0 units in-service | blog.192.168.50.4.nip.io |
+-------------+-------------------------+--------------------------+
```

You can then send the code of your application.

4.3.3 Application code

This document will not focus on how to write a Django blog, you can clone the entire source direct from GitHub: https://github.com/tsuru/tsuru-django-sample. Here is what we did for the project:

1. Create the project (`django-admin.py startproject`)
2. Enable django-admin
3. Install South
4. Create a "posts" app (`django-admin.py startapp posts`)
5. Add a "Post" model to the app
6. Register the model in django-admin
7. Generate the migration using South

4.3.4 Git deployment

When you create a new app, tsuru will display the Git remote that you should use. You can always get it using the command *app-info*:

```
$ tsuru app-info --app blog
Application: blog
Repository: git@192.168.50.4.nip.io:blog.git
Platform: python
Teams: admin
Address: blog.192.168.50.4.nip.io
Owner: admin@example.com
Team owner: admin
Deploys: 0
Pool: theonepool

App Plan:
+---------------+--------+------+-----------+--------+---------+
| Name          | Memory | Swap | Cpu Share | Router | Default |
+---------------+--------+------+-----------+--------+---------+
```

```
| autogenerated | 0 MB    | 0 MB | 100       |         | false    |
+---------------+---------+------+-----------+---------+----------+
```

The Git remote will be used to deploy your application using Git. You can just push to tsuru remote and your project will be deployed:

```
$ git push git@192.168.50.4.nip.io:blog.git master
Counting objects: 119, done.
Delta compression using up to 4 threads.
Compressing objects: 100% (53/53), done.
Writing objects: 100% (119/119), 16.24 KiB, done.
Total 119 (delta 55), reused 119 (delta 55)
remote:
remote:  ---> tsuru receiving push
remote:
remote: From git://cloud.tsuru.io/blog.git
remote:  * branch            master     -> FETCH_HEAD
remote:
remote:  ---> Installing dependencies
####################################
#          OMIT (see below)        #
####################################
remote:  ---> Restarting your app
remote:
remote:  ---> Deploy done!
remote:
To git@192.168.50.4.nip.io:blog.git
   a211fba..bbf5b53  master -> master
```

If you get a "Permission denied (publickey).", make sure you're member of a team and have a public key added to tsuru. To add a key, use the command *key-add*:

```
$ tsuru key-add mykey ~/.ssh/id_rsa.pub
```

You can use `git remote add` to avoid typing the entire remote url every time you want to push:

```
$ git remote add tsuru git@192.168.50.4.nip.io:blog.git
```

Then you can run:

```
$ git push tsuru master
Everything up-to-date
```

And you will be also able to omit the `--app` flag from now on:

```
$ tsuru app-info
Application: blog
Repository: git@192.168.50.4.nip.io:blog.git
Platform: python
Teams: admin
Address: blog.192.168.50.4.nip.io
Owner: admin@example.com
Team owner: admin
Deploys: 0
Pool: theonepool
Units: 1
+------------+---------+
| Unit       | State   |
+------------+---------+
| eab5151eff | started |
```

```
+------------+---------+

App Plan:
+---------------+--------+------+-----------+--------+---------+
| Name          | Memory | Swap | Cpu Share | Router | Default |
+---------------+--------+------+-----------+--------+---------+
| autogenerated | 0 MB   | 0 MB | 100       |        | false   |
+---------------+--------+------+-----------+--------+---------+
```

4.3.5 Listing dependencies

In the last section we omitted the dependencies step of deploy. In tsuru, an application can have two kinds of dependencies:

- **Operating system dependencies**, represented by packages in the package manager of the underlying operating system (e.g.: `yum` and `apt-get`);
- **Platform dependencies**, represented by packages in the package manager of the platform/language (in Python, `pip`).

All `apt-get` dependencies must be specified in a `requirements.apt` file, located in the root of your application, and pip dependencies must be located in a file called `requirements.txt`, also in the root of the application. Since we will use MySQL with Django, we need to install `mysql-python` package using `pip`, and this package depends on two `apt-get` packages: `python-dev` and `libmysqlclient-dev`, so here is how `requirements.apt` looks like:

```
libmysqlclient-dev
python-dev
```

And here is `requirements.txt`:

```
Django==1.4.1
MySQL-python==1.2.3
South==0.7.6
```

Please notice that we've included `South` too, for database migrations, and `Django`, off-course.

You can see the complete output of installing these dependencies below:

```
% git push tsuru master
####################################
#               OMIT               #
####################################
remote: Reading package lists...
remote: Building dependency tree...
remote: Reading state information...
remote: python-dev is already the newest version.
remote: The following extra packages will be installed:
remote:   libmysqlclient18 mysql-common
remote: The following NEW packages will be installed:
remote:   libmysqlclient-dev libmysqlclient18 mysql-common
remote: 0 upgraded, 3 newly installed, 0 to remove and 0 not upgraded.
remote: Need to get 2360 kB of archives.
remote: After this operation, 9289 kB of additional disk space will be used.
remote: Get:1 http://archive.ubuntu.com/ubuntu/ quantal/main mysql-common all 5.5.27-0ubuntu2 [13.7 k
remote: Get:2 http://archive.ubuntu.com/ubuntu/ quantal/main libmysqlclient18 amd64 5.5.27-0ubuntu2
remote: Get:3 http://archive.ubuntu.com/ubuntu/ quantal/main libmysqlclient-dev amd64 5.5.27-0ubuntu2
remote: debconf: unable to initialize frontend: Dialog
remote: debconf: (Dialog frontend will not work on a dumb terminal, an emacs shell buffer, or without
```

```
remote: debconf: falling back to frontend: Readline
remote: debconf: unable to initialize frontend: Readline
remote: debconf: (This frontend requires a controlling tty.)
remote: debconf: falling back to frontend: Teletype
remote: dpkg-preconfigure: unable to re-open stdin:
remote: Fetched 2360 kB in 1s (1285 kB/s)
remote: Selecting previously unselected package mysql-common.
remote: (Reading database ... 23143 files and directories currently installed.)
remote: Unpacking mysql-common (from .../mysql-common_5.5.27-0ubuntu2_all.deb) ...
remote: Selecting previously unselected package libmysqlclient18:amd64.
remote: Unpacking libmysqlclient18:amd64 (from .../libmysqlclient18_5.5.27-0ubuntu2_amd64.deb) ...
remote: Selecting previously unselected package libmysqlclient-dev.
remote: Unpacking libmysqlclient-dev (from .../libmysqlclient-dev_5.5.27-0ubuntu2_amd64.deb) ...
remote: Setting up mysql-common (5.5.27-0ubuntu2) ...
remote: Setting up libmysqlclient18:amd64 (5.5.27-0ubuntu2) ...
remote: Setting up libmysqlclient-dev (5.5.27-0ubuntu2) ...
remote: Processing triggers for libc-bin ...
remote: ldconfig deferred processing now taking place
remote: sudo: Downloading/unpacking Django==1.4.1 (from -r /home/application/current/requirements.txt
remote:   Running setup.py egg_info for package Django
remote:
remote: Downloading/unpacking MySQL-python==1.2.3 (from -r /home/application/current/requirements.txt
remote:   Running setup.py egg_info for package MySQL-python
remote:
remote:     warning: no files found matching 'MANIFEST'
remote:     warning: no files found matching 'ChangeLog'
remote:     warning: no files found matching 'GPL'
remote: Downloading/unpacking South==0.7.6 (from -r /home/application/current/requirements.txt (line
remote:   Running setup.py egg_info for package South
remote:
remote: Installing collected packages: Django, MySQL-python, South
remote:   Running setup.py install for Django
remote:     changing mode of build/scripts-2.7/django-admin.py from 644 to 755
remote:
remote:     changing mode of /usr/local/bin/django-admin.py to 755
remote:   Running setup.py install for MySQL-python
remote:     building '_mysql' extension
remote:     gcc -pthread -fno-strict-aliasing -DNDEBUG -g -fwrapv -O2 -Wall -Wstrict-prototypes -fPIC
remote:     In file included from _mysql.c:36:0:
remote:     /usr/include/mysql/my_config.h:422:0: warning: "HAVE_WCSCOLL" redefined [enabled by defau
remote:     In file included from /usr/include/python2.7/Python.h:8:0,
remote:                      from pymemcompat.h:10,
remote:                      from _mysql.c:29:
remote:     /usr/include/python2.7/pyconfig.h:890:0: note: this is the location of the previous defin
remote:     gcc -pthread -shared -Wl,-O1 -Wl,-Bsymbolic-functions -Wl,-Bsymbolic-functions -Wl,-z,rel
remote:
remote:     warning: no files found matching 'MANIFEST'
remote:     warning: no files found matching 'ChangeLog'
remote:     warning: no files found matching 'GPL'
remote:   Running setup.py install for South
remote:
remote: Successfully installed Django MySQL-python South
remote: Cleaning up...
####################################
#                OMIT              #
####################################
To git@192.168.50.4.nip.io:blog.git
   a211fba..bbf5b53  master -> master
```

4.3.6 Running the application

As you can see, in the deploy output there is a step described as "Restarting your app". In this step, tsuru will restart your app if it's running, or start it if it's not. But how does tsuru start an application? That's very simple, it uses a Procfile (a concept stolen from Foreman). In this Procfile, you describe how your application should be started. We can use gunicorn, for example, to start our Django application. Here is how the Procfile should look like:

```
web: gunicorn -b 0.0.0.0:$PORT blog.wsgi
```

Now we commit the file and push the changes to tsuru git server, running another deploy:

```
$ git add Procfile
$ git commit -m "Procfile: added file"
$ git push tsuru master
Counting objects: 5, done.
Delta compression using up to 4 threads.
Compressing objects: 100% (2/2), done.
Writing objects: 100% (3/3), 326 bytes, done.
Total 3 (delta 1), reused 0 (delta 0)
remote:
remote:  ---> tsuru receiving push
remote:
remote:  ---> Installing dependencies
remote: Reading package lists...
remote: Building dependency tree...
remote: Reading state information...
remote: python-dev is already the newest version.
remote: libmysqlclient-dev is already the newest version.
remote: 0 upgraded, 0 newly installed, 0 to remove and 1 not upgraded.
remote: Requirement already satisfied (use --upgrade to upgrade): Django==1.4.1 in /usr/local/lib/pyt
remote: Requirement already satisfied (use --upgrade to upgrade): MySQL-python==1.2.3 in /usr/local/l
remote: Requirement already satisfied (use --upgrade to upgrade): South==0.7.6 in /usr/local/lib/pyth
remote: Cleaning up...
remote:
remote:  ---> Restarting your app
remote: /var/lib/tsuru/hooks/start: line 13: gunicorn: command not found
remote:
remote:  ---> Deploy done!
remote:
To git@192.168.50.4.nip.io:blog.git
   81e884e..530c528  master -> master
```

Now we get an error: `gunicorn: command not found`. It means that we need to add gunicorn to `requirements.txt` file:

```
$ cat >> requirements.txt
gunicorn==0.14.6
^D
```

Now we commit the changes and run another deploy:

```
$ git add requirements.txt
$ git commit -m "requirements.txt: added gunicorn"
$ git push tsuru master
Counting objects: 5, done.
Delta compression using up to 4 threads.
Compressing objects: 100% (3/3), done.
Writing objects: 100% (3/3), 325 bytes, done.
Total 3 (delta 1), reused 0 (delta 0)
```

```
remote:
remote:  ---> tsuru receiving push
remote:
[...]
remote:  ---> Restarting your app
remote:
remote:  ---> Deploy done!
remote:
To git@192.168.50.4.nip.io:blog.git
   530c528..542403a  master -> master
```

Now that the app is deployed, you can access it from your browser, getting the IP or host listed in `app-list` and opening it. For example, in the list below:

```
$ tsuru app-list
+-------------+------------------------+---------------------+
| Application | Units State Summary    | Address             |
+-------------+------------------------+---------------------+
| blog        | 1 of 1 units in-service | blog.cloud.tsuru.io |
+-------------+------------------------+---------------------+
```

We can access the admin of the app in the URL http://blog.cloud.tsuru.io/admin/.

4.3.7 Using services

Now that gunicorn is running, we can accesss the application in the browser, but we get a Django error: *"Can't connect to local MySQL server through socket '/var/run/mysqld/mysqld.sock' (2)"*. This error means that we can't connect to MySQL on localhost. That's because we should not connect to MySQL on localhost, we must use a service. The service workflow can be resumed to two steps:

1. Create a service instance
2. Bind the service instance to the app

But how can I see what services are available? Easy! Use the command *service-list*:

```
$ tsuru service-list
+---------------+-----------+
| Services      | Instances |
+---------------+-----------+
| elastic-search |           |
| mysql         |           |
+---------------+-----------+
```

The output from `service-list` above says that there are two available services: "elastic-search" and "mysql", and no instances. To create our MySQL instance, we should run the command *service-add*:

```
$ tsuru service-add mysql blogsql
Service successfully added.
```

Now, if we run `service-list` again, we will see our new service instance in the list:

```
$ tsuru service-list
+---------------+-----------+
| Services      | Instances |
+---------------+-----------+
| elastic-search |           |
| mysql         | blogsql   |
+---------------+-----------+
```

To bind the service instance to the application, we use the command *service-bind*:

```
$ tsuru service-bind blogsql
Instance blogsql is now bound to the app blog.

The following environment variables are now available for use in your app:

- MYSQL_PORT
- MYSQL_PASSWORD
- MYSQL_USER
- MYSQL_HOST
- MYSQL_DATABASE_NAME

For more details, please check the documentation for the service, using service-doc command.
```

As you can see from bind output, we use environment variables to connect to the MySQL server. Next step is update `settings.py` to use these variables to connect in the database:

```
import os

DATABASES = {
    'default': {
        'ENGINE': 'django.db.backends.mysql',
        'NAME': os.environ.get('MYSQL_DATABASE_NAME', 'blog'),
        'USER': os.environ.get('MYSQL_USER', 'root'),
        'PASSWORD': os.environ.get('MYSQL_PASSWORD', ''),
        'HOST': os.environ.get('MYSQL_HOST', ''),
        'PORT': os.environ.get('MYSQL_PORT', ''),
    }
}
```

Now let's commit it and run another deploy:

```
$ git add blog/settings.py
$ git commit -m "settings: using environment variables to connect to MySQL"
$ git push tsuru master
Counting objects: 7, done.
Delta compression using up to 4 threads.
Compressing objects: 100% (4/4), done.
Writing objects: 100% (4/4), 535 bytes, done.
Total 4 (delta 3), reused 0 (delta 0)
remote:
remote:  ---> tsuru receiving push
remote:
remote:  ---> Installing dependencies
####################################
#               OMIT               #
####################################
remote:
remote:  ---> Restarting your app
remote:
remote:  ---> Deploy done!
remote:
To git@192.168.50.4.nip.io:blog.git
   ab4e706..a780de9  master -> master
```

Now if we try to access the admin again, we will get another error: *"Table 'blogsql.django_session' doesn't exist"*. Well, that means that we have access to the database, so bind worked, but we did not set up the database yet. We need to run `syncdb` and `migrate` (if we're using South) in the remote server. We can use the command *app-run*, we could write:

```
$ tsuru app-run -- python manage.py syncdb --noinput
Syncing...
Creating tables ...
Creating table auth_permission
Creating table auth_group_permissions
Creating table auth_group
Creating table auth_user_user_permissions
Creating table auth_user_groups
Creating table auth_user
Creating table django_content_type
Creating table django_session
Creating table django_site
Creating table django_admin_log
Creating table south_migrationhistory
Installing custom SQL ...
Installing indexes ...
Installed 0 object(s) from 0 fixture(s)

Synced:
 > django.contrib.auth
 > django.contrib.contenttypes
 > django.contrib.sessions
 > django.contrib.sites
 > django.contrib.messages
 > django.contrib.staticfiles
 > django.contrib.admin
 > south

Not synced (use migrations):
 - blog.posts
(use ./manage.py migrate to migrate these)
```

The same applies for `migrate`.

4.3.8 Deployment hooks

It would be boring to manually run `syncdb` and/or `migrate` after every deployment. So we can configure an automatic hook to always run before or after the app restarts.

tsuru parses a file called `tsuru.yaml` and runs restart hooks. As the extension suggests, this is a YAML file, that contains a list of commands that should run before and after the restart. Here is our example of tsuru.yaml:

```
hooks:
  build:
    - python manage.py syncdb --noinput
    - python manage.py migrate
```

For more details, check the *hooks documentation*.

tsuru will look for the file in the root of the project. Let's commit and deploy it:

```
$ git add tsuru.yaml
$ git commit -m "tsuru.yaml: added file"
$ git push tsuru master
Counting objects: 4, done.
Delta compression using up to 4 threads.
Compressing objects: 100% (3/3), done.
Writing objects: 100% (3/3), 338 bytes, done.
```

```
Total 3 (delta 1), reused 0 (delta 0)
remote:
remote:  ---> tsuru receiving push
remote:
remote:  ---> Installing dependencies
remote: Reading package lists...
remote: Building dependency tree...
remote: Reading state information...
remote: python-dev is already the newest version.
remote: libmysqlclient-dev is already the newest version.
remote: 0 upgraded, 0 newly installed, 0 to remove and 15 not upgraded.
remote: Requirement already satisfied (use --upgrade to upgrade): Django==1.4.1 in /usr/local/lib/pyt
remote: Requirement already satisfied (use --upgrade to upgrade): MySQL-python==1.2.3 in /usr/local/l
remote: Requirement already satisfied (use --upgrade to upgrade): South==0.7.6 in /usr/local/lib/pyth
remote: Requirement already satisfied (use --upgrade to upgrade): gunicorn==0.14.6 in /usr/local/lib/
remote: Cleaning up...
remote:
remote:  ---> Restarting your app
remote:
remote:  ---> Running restart:after
remote:
remote:  ---> Deploy done!
remote:
To git@192.168.50.4.nip.io:blog.git
   a780de9..1b675b8  master -> master
```

It's done! Now we have a Django project deployed on tsuru, using a MySQL service.

4.3.9 Going further

For more information, you can dig into tsuru docs, or read complete instructions of use for the tsuru command.

4.4 Deploying Ruby applications in tsuru

4.4.1 Overview

This document is a hands-on guide to deploying a simple Ruby application in tsuru. The example application will be a very simple Rails project associated to a MySQL service.

4.4.2 Creating the app within tsuru

To create an app, you use the command *app-create*:

```
$ tsuru app-create <app-name> <app-platform>
```

For Ruby, the app platform is `ruby`! Let's be over creative and develop a never-developed tutorial-app: a blog, and its name will also be very creative, let's call it "blog":

```
$ tsuru app-create blog ruby
```

To list all available platforms, use the command *platform-list*.

You can see all your applications using the command *app-list*:

```
$ tsuru app-list
+-------------+-------------------------+-------------+
| Application | Units State Summary     | Address     |
+-------------+-------------------------+-------------+
| blog        | 0 of 0 units in-service |             |
+-------------+-------------------------+-------------+
```

4.4.3 Application code

This document will not focus on how to write a blog with Rails, you can clone the entire source direct from GitHub: https://github.com/tsuru/tsuru-ruby-sample. Here is what we did for the project:

1. Create the project (`rails new blog`)
2. Generate the scaffold for Post (`rails generate scaffold Post title:string body:text`)

4.4.4 Git deployment

When you create a new app, tsuru will display the Git remote that you should use. You can always get it using the command *app-info*:

```
$ tsuru app-info --app blog
Application: blog
Repository: git@192.168.50.4.nip.io:blog.git
Platform: ruby
Teams: admin
Address: blog.192.168.50.4.nip.io
Owner: admin@example.com
Team owner: admin
Deploys: 0
Pool: theonepool

App Plan:
+---------------+--------+------+-----------+--------+---------+
| Name          | Memory | Swap | Cpu Share | Router | Default |
+---------------+--------+------+-----------+--------+---------+
| autogenerated | 0 MB   | 0 MB | 100       |        | false   |
+---------------+--------+------+-----------+--------+---------+
```

The Git remote will be used to deploy your application using Git. You can just push to tsuru remote and your project will be deployed:

```
$ git push git@192.168.50.4.nip.io:blog.git master
Counting objects: 86, done.
Delta compression using up to 4 threads.
Compressing objects: 100% (75/75), done.
Writing objects: 100% (86/86), 29.75 KiB, done.
Total 86 (delta 2), reused 0 (delta 0)
remote: Cloning into '/home/application/current'...
remote: requirements.apt not found.
remote: Skipping...
remote: /home/application/current /
remote: Fetching gem metadata from https://rubygems.org/.........
remote: Fetching gem metadata from https://rubygems.org/..
###################################
#          OMIT (see below)         #
```

```
####################################
remote:  ---> App will be restarted, please check its log for more details...
remote:
To git@192.168.50.4.nip.io:blog.git
 * [new branch]      master -> master
```

If you get a "Permission denied (publickey).", make sure you're member of a team and have a public key added to tsuru. To add a key, use the command *key-add*:

```
$ tsuru key-add mykey ~/.ssh/id_rsa.pub
```

You can use `git remote add` to avoid typing the entire remote url every time you want to push:

```
$ git remote add tsuru git@192.168.50.4.nip.io:blog.git
```

Then you can run:

```
$ git push tsuru master
Everything up-to-date
```

And you will be also able to omit the `--app` flag from now on:

```
$ tsuru app-info
Application: blog
Repository: git@192.168.50.4.nip.io:blog.git
Platform: ruby
Teams: admin
Address: blog.192.168.50.4.nip.io
Owner: admin@example.com
Team owner: admin
Deploys: 0
Pool: theonepool
Units: 1
+------------+---------+
| Unit       | State   |
+------------+---------+
| eab5151eff | started |
+------------+---------+

App Plan:
+---------------+--------+------+-----------+--------+---------+
| Name          | Memory | Swap | Cpu Share | Router | Default |
+---------------+--------+------+-----------+--------+---------+
| autogenerated | 0 MB   | 0 MB | 100       |        | false   |
+---------------+--------+------+-----------+--------+---------+
```

4.4.5 Listing dependencies

In the last section we omitted the dependencies step of deploy. In tsuru, an application can have two kinds of dependencies:

- **Operating system dependencies**, represented by packages in the package manager of the underlying operating system (e.g.: `yum` and `apt-get`);
- **Platform dependencies**, represented by packages in the package manager of the platform/language (in Ruby, `bundler`).

All `apt-get` dependencies must be specified in a `requirements.apt` file, located in the root of your application, and ruby dependencies must be located in a file called `Gemfile`, also in the root of the application. Since we will

use MySQL with Rails, we need to install `mysql` package using `gem`, and this package depends on an `apt-get` package: `libmysqlclient-dev`, so here is how `requirements.apt` looks like:

```
libmysqlclient-dev
```

And here is `Gemfile`:

```
source 'https://rubygems.org'

gem 'rails', '3.2.13'
gem 'mysql'
gem 'sass-rails',   '~> 3.2.3'
gem 'coffee-rails', '~> 3.2.1'
gem 'therubyracer', platforms: 'ruby'
gem 'uglifier', '>= 1.0.3'
gem 'jquery-rails'
```

You can see the complete output of installing these dependencies below:

```
$ git push tsuru master
####################################
#                OMIT              #
####################################
remote: Reading package lists...
remote: Building dependency tree...
remote: Reading state information...
remote: The following extra packages will be installed:
remote:   libmysqlclient18 mysql-common
remote: The following NEW packages will be installed:
remote:   libmysqlclient-dev libmysqlclient18 mysql-common
remote: 0 upgraded, 3 newly installed, 0 to remove and 0 not upgraded.
remote: Need to get 2360 kB of archives.
remote: After this operation, 9289 kB of additional disk space will be used.
remote: Get:1 http://archive.ubuntu.com/ubuntu/ quantal/main mysql-common all 5.5.27-0ubuntu2 [13.7 k
remote: Get:2 http://archive.ubuntu.com/ubuntu/ quantal/main libmysqlclient18 amd64 5.5.27-0ubuntu2
remote: Get:3 http://archive.ubuntu.com/ubuntu/ quantal/main libmysqlclient-dev amd64 5.5.27-0ubuntu2
remote: Fetched 2360 kB in 2s (1112 kB/s)
remote: Selecting previously unselected package mysql-common.
remote: (Reading database ... 41063 files and directories currently installed.)
remote: Unpacking mysql-common (from .../mysql-common_5.5.27-0ubuntu2_all.deb) ...
remote: Selecting previously unselected package libmysqlclient18:amd64.
remote: Unpacking libmysqlclient18:amd64 (from .../libmysqlclient18_5.5.27-0ubuntu2_amd64.deb) ...
remote: Selecting previously unselected package libmysqlclient-dev.
remote: Unpacking libmysqlclient-dev (from .../libmysqlclient-dev_5.5.27-0ubuntu2_amd64.deb) ...
remote: Setting up mysql-common (5.5.27-0ubuntu2) ...
remote: Setting up libmysqlclient18:amd64 (5.5.27-0ubuntu2) ...
remote: Setting up libmysqlclient-dev (5.5.27-0ubuntu2) ...
remote: Processing triggers for libc-bin ...
remote: ldconfig deferred processing now taking place
remote: /home/application/current /
remote: Fetching gem metadata from https://rubygems.org/.........
remote: Fetching gem metadata from https://rubygems.org/..
remote: Using rake (10.1.0)
remote: Using i18n (0.6.1)
remote: Using multi_json (1.7.8)
remote: Using activesupport (3.2.13)
remote: Using builder (3.0.4)
remote: Using activemodel (3.2.13)
remote: Using erubis (2.7.0)
```

```
remote: Using journey (1.0.4)
remote: Using rack (1.4.5)
remote: Using rack-cache (1.2)
remote: Using rack-test (0.6.2)
remote: Using hike (1.2.3)
remote: Using tilt (1.4.1)
remote: Using sprockets (2.2.2)
remote: Using actionpack (3.2.13)
remote: Using mime-types (1.23)
remote: Using polyglot (0.3.3)
remote: Using treetop (1.4.14)
remote: Using mail (2.5.4)
remote: Using actionmailer (3.2.13)
remote: Using arel (3.0.2)
remote: Using tzinfo (0.3.37)
remote: Using activerecord (3.2.13)
remote: Using activeresource (3.2.13)
remote: Using coffee-script-source (1.6.3)
remote: Using execjs (1.4.0)
remote: Using coffee-script (2.2.0)
remote: Using rack-ssl (1.3.3)
remote: Using json (1.8.0)
remote: Using rdoc (3.12.2)
remote: Using thor (0.18.1)
remote: Using railties (3.2.13)
remote: Using coffee-rails (3.2.2)
remote: Using jquery-rails (3.0.4)
remote: Installing libv8 (3.11.8.17)
remote: Installing mysql (2.9.1)
remote: Using bundler (1.3.5)
remote: Using rails (3.2.13)
remote: Installing ref (1.0.5)
remote: Using sass (3.2.10)
remote: Using sass-rails (3.2.6)
remote: Installing therubyracer (0.11.4)
remote: Installing uglifier (2.1.2)
remote: Your bundle is complete!
remote: Gems in the groups test and development were not installed.
remote: It was installed into ./vendor/bundle
####################################
#               OMIT               #
####################################
To git@192.168.50.4.nip.io:blog.git
   9515685..d67c3cd  master -> master
```

4.4.6 Running the application

As you can see, in the deploy output there is a step described as "Restarting your app". In this step, tsuru will restart your app if it's running, or start it if it's not. But how does tsuru start an application? That's very simple, it uses a Procfile (a concept stolen from Foreman). In this Procfile, you describe how your application should be started. Here is how the Procfile should look like:

```
web: bundle exec rails server -p $PORT -e production
```

Now we commit the file and push the changes to tsuru Git server, running another deploy:

```
$ git add Procfile
$ git commit -m "Procfile: added file"
$ git push tsuru master
####################################
#                OMIT              #
####################################
remote:  ---> App will be restarted, please check its log for more details...
remote:
To git@192.168.50.4.nip.io:blog.git
   d67c3cd..f2a5d2d  master -> master
```

Now that the app is deployed, you can access it from your browser, getting the IP or host listed in `app-list` and opening it. For example, in the list below:

```
$ tsuru app-list
+-------------+-------------------------+---------------------+
| Application | Units State Summary     | Address             |
+-------------+-------------------------+---------------------+
| blog        | 1 of 1 units in-service | blog.cloud.tsuru.io |
+-------------+-------------------------+---------------------+
```

4.4.7 Using services

Now that your app is not running with success because the rails can't connect to MySQL. That's because we add a relation between your rails app and a mysql instance. To do it we must use a service. The service workflow can be resumed to two steps:

1. Create a service instance
2. Bind the service instance to the app

But how can I see what services are available? Easy! Use the command *service-list*:

```
$ tsuru service-list
+----------------+-----------+
| Services       | Instances |
+----------------+-----------+
| elastic-search |           |
| mysql          |           |
+----------------+-----------+
```

The output from `service-list` above says that there are two available services: "elastic-search" and "mysql", and no instances. To create our MySQL instance, we should run the command *service-add*:

```
$ tsuru service-add mysql blogsql
Service successfully added.
```

Now, if we run `service-list` again, we will see our new service instance in the list:

```
$ tsuru service-list
+----------------+-----------+
| Services       | Instances |
+----------------+-----------+
| elastic-search |           |
| mysql          | blogsql   |
+----------------+-----------+
```

To bind the service instance to the application, we use the command *service-bind*:

```
$ tsuru service-bind blogsql
Instance blogsql is now bound to the app blog.

The following environment variables are now available for use in your app:

- MYSQL_PORT
- MYSQL_PASSWORD
- MYSQL_USER
- MYSQL_HOST
- MYSQL_DATABASE_NAME

For more details, please check the documentation for the service, using service-doc command.
```

As you can see from bind output, we use environment variables to connect to the MySQL server. Next step is update `conf/database.yml` to use these variables to connect in the database:

```
production:
  adapter: mysql
  encoding: utf8
  database: <%= ENV["MYSQL_DATABASE_NAME"] %>
  pool: 5
  username: <%= ENV["MYSQL_USER"] %>
  password: <%= ENV["MYSQL_PASSWORD"] %>
  host: <%= ENV["MYSQL_HOST"] %>
```

Now let's commit it and run another deploy:

```
$ git add conf/database.yml
$ git commit -m "database.yml: using environment variables to connect to MySQL"
$ git push tsuru master
Counting objects: 7, done.
Delta compression using up to 4 threads.
Compressing objects: 100% (4/4), done.
Writing objects: 100% (4/4), 535 bytes, done.
Total 4 (delta 3), reused 0 (delta 0)
remote:
remote:  ---> tsuru receiving push
remote:
remote:  ---> Installing dependencies
####################################
#               OMIT               #
####################################
remote:
remote:  ---> Restarting your app
remote:
remote:  ---> Deploy done!
remote:
To git@192.168.50.4.nip.io:blog.git
   ab4e706..a780de9  master -> master
```

Now if we try to access the admin again, we will get another error: *"Table 'blogsql.django_session' doesn't exist"*. Well, that means that we have access to the database, so bind worked, but we did not set up the database yet. We need to run `rake db:migrate` in the remote server. We can use the command *app-run* to execute commands in the machine, so for running `rake db:migrate` we could write:

```
$ tsuru app-run -- RAILS_ENV=production bundle exec rake db:migrate
==  CreatePosts: migrating ============================================
-- create_table(:posts)
   -> 0.1126s
```

```
==  CreatePosts: migrated (0.1128s) ============================================
```

4.4.8 Deployment hooks

It would be boring to manually run `rake db:migrate` after every deployment. So we can configure an automatic hook to always run before or after the app restarts.

tsuru parses a file called `tsuru.yaml` and runs restart hooks. As the extension suggests, this is a YAML file, that contains a list of commands that should run before and after the restart. Here is our example of tsuru.yaml:

```
hooks:
  restart:
    before-each:
      - RAILS_ENV=production bundle exec rake db:migrate
```

For more details, check the *hooks documentation*.

tsuru will look for the file in the root of the project. Let's commit and deploy it:

```
$ git add tsuru.yaml
$ git commit -m "tsuru.yaml: added file"
$ git push tsuru master
####################################
#                OMIT              #
####################################
To git@192.168.50.4.nip.io:blog.git
   a780de9..1b675b8  master -> master
```

It is necessary to compile de assets before the app restart. To do it we can use the `rake assets:precompile` command. Then let's add the command to compile the assets in tsuru.yaml:

```
hooks:
  build:
    - RAILS_ENV=production bundle exec rake assets:precompile
```

```
$ git add tsuru.yaml
$ git commit -m "tsuru.yaml: added file"
$ git push tsuru master
####################################
#                OMIT              #
####################################
To git@192.168.50.4.nip.io:blog.git
   a780de9..1b675b8  master -> master
```

It's done! Now we have a Rails project deployed on tsuru, using a MySQL service.

Now we can access your *blog app* in the URL returned in *app-info*.

4.4.9 Going further

For more information, you can dig into the tsuru docs, or read the complete instructions on how to use the tsuru command.

4.5 Deploying Go applications in tsuru

4.5.1 Overview

This document is a hands-on guide to deploying a simple Go web application in tsuru.

4.5.2 Creating the app within tsuru

To create an app, you use the command *app-create*:

```
$ tsuru app-create <app-name> <app-platform>
```

For Go, the platform name is `go`! Let's be over creative and develop a hello world tutorial-app, let's call it "helloworld":

```
$ tsuru app-create helloworld go
```

To list all available platforms, use the command *platform-list*.

You can see all your applications using the command *app-list*:

```
$ tsuru app-list
+-------------+-------------------------+---------------------------------+
| Application | Units State Summary     | Address                         |
+-------------+-------------------------+---------------------------------+
| helloworld  | 0 of 0 units in-service | helloworld.192.168.50.4.nip.io |
+-------------+-------------------------+---------------------------------+
```

4.5.3 Application code

A simple web application in Go *main.go*:

```
package main

import (
    "fmt"
    "net/http"
    "os"
)

func main() {
    http.HandleFunc("/", hello)
    fmt.Println("listening...")
    err := http.ListenAndServe(":" + os.Getenv("PORT"), nil)
    if err != nil {
        panic(err)
    }
}

func hello(res http.ResponseWriter, req *http.Request) {
    fmt.Fprintln(res, "hello, world!")
}
```

4.5.4 Git deployment

When you create a new app, tsuru will display the Git remote that you should use. You can always get it using the command *app-info*:

```
$ tsuru app-info --app helloworld
Application: helloworld
Repository: git@192.168.50.4.nip.io:helloworld.git
Platform: go
Teams: admin
Address: helloworld.192.168.50.4.nip.io
Owner: admin@example.com
Team owner: admin
Deploys: 0
Pool: theonepool

App Plan:
+---------------+--------+------+-----------+--------+---------+
| Name          | Memory | Swap | Cpu Share | Router | Default |
+---------------+--------+------+-----------+--------+---------+
| autogenerated | 0 MB   | 0 MB | 100       |        | false   |
+---------------+--------+------+-----------+--------+---------+
```

The git remote will be used to deploy your application using git. You can just push to tsuru remote and your project will be deployed:

```
$ git push git@192.168.50.4.nip.io:helloworld.git master
Counting objects: 3, done.
Delta compression using up to 4 threads.
Compressing objects: 100% (2/2), done.
Writing objects: 100% (3/3), 430 bytes | 0 bytes/s, done.
Total 3 (delta 0), reused 0 (delta 0)
remote: tar: Removing leading `/' from member names
remote: /
remote:
remote: ---- Building application image ----
remote:  ---> Sending image to repository (5.57MB)
remote:  ---> Cleaning up
remote:
remote: ---- Starting 1 new unit ----
remote:  ---> Started unit b21298a64e...
remote:
remote: ---- Binding and checking 1 new units ----
remote:  ---> Bound and checked unit b21298a64e
remote:
remote: ---- Adding routes to 1 new units ----
remote:  ---> Added route to unit b21298a64e
remote:
remote: OK
To git@192.168.50.4.nip.io:helloworld.git
 * [new branch]      master -> master
```

If you get a "Permission denied (publickey).", make sure you're member of a team and have a public key added to tsuru. To add a key, use the command *key-add*:

```
$ tsuru key-add mykey ~/.ssh/id_rsa.pub
```

You can use `git remote add` to avoid typing the entire remote url every time you want to push:

```
$ git remote add tsuru git@192.168.50.4.nip.io:helloworld.git
```

Then you can run:

```
$ git push tsuru master
Everything up-to-date
```

And you will be also able to omit the `--app` flag from now on:

```
$ tsuru app-info
Application: helloworld
Repository: git@192.168.50.4.nip.io:helloworld.git
Platform: go
Teams: admin
Address: helloworld.192.168.50.4.nip.io
Owner: admin@example.com
Team owner: admin
Deploys: 1
Pool: theonepool
Units: 1
+------------+---------+
| Unit       | State   |
+------------+---------+
| b21298a64e | started |
+------------+---------+

App Plan:
+---------------+--------+------+-----------+--------+---------+
| Name          | Memory | Swap | Cpu Share | Router | Default |
+---------------+--------+------+-----------+--------+---------+
| autogenerated | 0 MB   | 0 MB | 100       |        | false   |
+---------------+--------+------+-----------+--------+---------+
```

4.5.5 Running the application

tsuru will compile and run the application automatically, but it's possible to customize how tsuru compiles and runs the application. For more details, check the README of the Go platform: https://github.com/tsuru/basebuilder/blob/master/go/README.md.

Now that the app is deployed, you can access it from your browser, getting the IP or host listed in `app-list` and opening it. For example, in the list below:

```
$ tsuru app-list
+-------------+-------------------------+--------------------------------+
| Application | Units State Summary     | Address                        |
+-------------+-------------------------+--------------------------------+
| helloworld  | 1 of 1 units in-service | helloworld.192.168.50.4.nip.io |
+-------------+-------------------------+--------------------------------+
```

It's done! Now we have a simple go project deployed on tsuru.

Now we can access your app in the URL displayed in *app-list* ("helloworld.192.168.50.4.nip.io" in this case).

4.5.6 Going further

For more information, you can dig into tsuru docs, or read complete instructions of use for the tsuru command.

4.6 Deploying Java applications on tsuru

4.6.1 Overview

This document is a hands-on guide to deploying a simple Java application on tsuru. The example application is a simple mvn generated archetype, in order to generate it, just run:

```
$ mvn archetype:generate -DgroupId=io.tsuru.javasample -DartifactId=helloweb -DarchetypeArtifactId=ma
```

You can also deploy any other Java application you have on a tsuru server. Another alternative is to just download the code available at GitHub: https://github.com/tsuru/tsuru-java-sample.

4.6.2 Creating the app within tsuru

To create an app, you use the command *app-create*:

```
$ tsuru app-create <app-name> <app-platform>
```

For Java, the app platform is, guess what, `java`! Let's call our application "helloweb":

```
$ tsuru app-create helloweb java
```

To list all available platforms, use the command *platform-list*.

You can see all your applications using the command *app-list*:

```
$ tsuru app-list
+-------------+-------------------------+------------------------------+
| Application | Units State Summary     | Address                      |
+-------------+-------------------------+------------------------------+
| helloweb    | 0 of 0 units in-service | helloweb.192.168.50.4.nip.io |
+-------------+-------------------------+------------------------------+
```

4.6.3 Deploying the code

Using the Java platform, there are two deployment strategies: users can either upload WAR files to tsuru or send the code using the regular `git push` approach. This guide will cover both approaches:

WAR deployment

Using the mvn archetype, generating the WAR is as easy as running `mvn package`, then the user can deploy the code using `tsuru app-deploy`:

```
$ mvn package
$ cd target
$ tsuru app-deploy -a helloweb helloweb.war
Uploading files.... ok

---- Building application image ----
 ---> Sending image to repository (0.00MB)
 ---> Cleaning up

---- Starting 1 new unit ----
 ---> Started unit 21c3b6aafa...
```

```
---- Binding and checking 1 new units ----
 ---> Bound and checked unit 21c3b6aafa

---- Adding routes to 1 new units ----
 ---> Added route to unit 21c3b6aafa

OK
```

Done! Now you can access your project in the address displayed in the output of *tsuru app-list*. Remeber to add `/helloweb/`.

You can also deploy you application to the / address, renaming the WAR to ROOT.war and redeploying it:

```
$ mv helloweb.war ROOT.war
$ tsuru app-deploy -a helloweb ROOT.war
Uploading files... ok

---- Building application image ----
 ---> Sending image to repository (0.00MB)
 ---> Cleaning up

---- Starting 1 new unit ----
 ---> Started unit 4d155e805f...

---- Adding routes to 1 new units ----
 ---> Added route to unit 4d155e805f

---- Removing routes from 1 old units ----
 ---> Removed route from unit d2811c0801

---- Removing 1 old unit ----
 ---> Removed old unit 1/1

OK
```

And now you can access your hello world in the root of the application address!

Git deployment

For Git deployment, we will send the code to tsuru, and compile the classes there. For that, we're going to use mvn with the Jetty plugin. For doing that, we will need to create a Procfile with the command for starting the application:

```
$ cat Procfile
helloweb: mvn jetty:run
```

In order to compile the application classes during deployment, we need also to add a deployment hook. tsuru parses a file called `tsuru.yaml` and runs some build hooks in the deployment phase.

Here is how the file for the `helloweb` application looks like:

```
$ cat tsuru.yaml
hooks:
  build:
    - mvn package
```

After adding these files, we're ready for deploying the application. The command *app-info* command will display a Git remote that we can use to push the application code to production:

```
$ tsuru app-info -a helloweb
Application: helloweb
Repository: git@192.168.50.4.nip.io:helloweb.git
Platform: java
Teams: admin
Address: helloweb.192.168.50.4.nip.io
Owner: admin@example.com
Team owner: admin
Deploys: 2
Pool: theonepool
Units: 1
+------------+---------+
| Unit       | State   |
+------------+---------+
| 313458bb9d | started |
+------------+---------+

App Plan:
+---------------+--------+------+-----------+--------+---------+
| Name          | Memory | Swap | Cpu Share | Router | Default |
+---------------+--------+------+-----------+--------+---------+
| autogenerated | 0 MB   | 0 MB | 100       |        | false   |
+---------------+--------+------+-----------+--------+---------+
```

The "Repository" line contains what we need: the remote repository. Now we can simply push the application code, using Git push:

```
$ git push git@192.168.50.4.nip.io:helloweb.git master
Counting objects: 25, done.
Delta compression using up to 4 threads.
Compressing objects: 100% (19/19), done.
Writing objects: 100% (25/25), 2.59 KiB | 0 bytes/s, done.
Total 25 (delta 5), reused 0 (delta 0)
remote: tar: Removing leading `/' from member names
remote: [INFO] Scanning for projects...
remote: [INFO]
remote: [INFO] ------------------------------------------------------------------------
remote: [INFO] Building helloweb Maven Webapp 1.0-SNAPSHOT
remote: [INFO] ------------------------------------------------------------------------
remote: Downloading: http://repo.maven.apache.org/maven2/org/apache/maven/plugins/maven-resources-plu
remote: Downloaded: http://repo.maven.apache.org/maven2/org/apache/maven/plugins/maven-resources-plug
remote: Downloading: http://repo.maven.apache.org/maven2/org/apache/maven/plugins/maven-plugins/12/ma
remote: Downloaded: http://repo.maven.apache.org/maven2/org/apache/maven/plugins/maven-plugins/12/mav

...

remote: [INFO] Packaging webapp
remote: [INFO] Assembling webapp [helloweb] in [/home/application/current/target/helloweb]
remote: [INFO] Processing war project
remote: [INFO] Copying webapp resources [/home/application/current/src/main/webapp]
remote: [INFO] Webapp assembled in [27 msecs]
remote: [INFO] Building war: /home/application/current/target/helloweb.war
remote: [INFO] WEB-INF/web.xml already added, skipping
remote: [INFO] ------------------------------------------------------------------------
remote: [INFO] BUILD SUCCESS
remote: [INFO] ------------------------------------------------------------------------
remote: [INFO] Total time: 51.729s
remote: [INFO] Finished at: Tue Nov 11 17:04:05 UTC 2014
```

```
remote: [INFO] Final Memory: 8M/19M
remote: [INFO] ---------------------------------------------------------------------
remote:
remote: ---- Building application image ----
remote:  ---> Sending image to repository (2.96MB)
remote:  ---> Cleaning up
remote:
remote: ---- Starting 1 new unit ----
remote:  ---> Started unit e71d176232...
remote:
remote: ---- Adding routes to 1 new units ----
remote:  ---> Added route to unit e71d176232
remote:
remote: ---- Removing routes from 1 old units ----
remote:  ---> Removed route from unit d8a2d14948
remote:
remote: ---- Removing 1 old unit ----
remote:  ---> Removed old unit 1/1
remote:
remote: OK
To git@tsuru.mycompany.com:helloweb.git
 * [new branch]      master -> master
```

As you can see, the final part of the output is the same, and the application is running in the address given by tsuru as well.

4.6.4 Switching between Java versions

In the Java platform provided by tsuru, users can use two version of Java: 7 and 8, both provided by Oracle. There's an environment variable for defining the Java version you wanna use: `JAVA_VERSION`. The default behavior of the platform is to use Java 7, but you can change to Java 8 by running:

```
$ tsuru env-set -a helloweb JAVA_VERSION=8
---- Setting 1 new environment variables ----

---- Starting 1 new unit ----
 ---> Started unit d8a2d14948...

---- Adding routes to 1 new units ----
 ---> Added route to unit d8a2d14948

---- Removing routes from 1 old units ----
 ---> Removed route from unit 4d155e805f

---- Removing 1 old unit ----
 ---> Removed old unit 1/1
```

And... done! No need to run another deployment, your application is now running with Java 8.

4.6.5 Going further

For more information, you can dig into tsuru docs, or read complete instructions of use for the tsuru command.

4.7 Deploying PHP applications in tsuru

4.7.1 Overview

This document is a hands-on guide to deploying a simple PHP application in tsuru. The example application will be a very simple Wordpress project associated to a MySQL service. It's applicable to any php over apache application.

4.7.2 Creating the app in tsuru

To create an app, you use the command *app-create*:

```
$ tsuru app-create <app-name> <app-platform>
```

For PHP, the app platform is, guess what, `php`! Let's be over creative and develop a never-developed tutorial-app: a blog, and its name will also be very creative, let's call it "blog":

```
$ tsuru app-create blog php
```

To list all available platforms, use the command *platform-list*.

You can see all your applications using the command *app-list*:

```
$ tsuru app-list
+-------------+-------------------------+--------------------------+
| Application | Units State Summary     | Address                  |
+-------------+-------------------------+--------------------------+
| blog        | 0 of 0 units in-service | blog.192.168.50.4.nip.io |
+-------------+-------------------------+--------------------------+
```

4.7.3 Application code

This document will not focus on how to write a php blog, you can download the entire source direct from wordpress: http://wordpress.org/latest.zip. Here is all you need to do with your project:

```
# Download and unpack wordpress
$ wget http://wordpress.org/latest.zip
$ unzip latest.zip
# Preparing wordpress for tsuru
$ cd wordpress
# Notify tsuru about the necessary packages
$ echo php5-mysql > requirements.apt
# Preparing the application to receive the tsuru environment related to the mysql service
$ sed "s/'database_name_here'/getenv('MYSQL_DATABASE_NAME')/; \
            s/'username_here'/getenv('MYSQL_USER')/; \
            s/'localhost'/getenv('MYSQL_HOST')/; \
            s/'password_here'/getenv('MYSQL_PASSWORD')/" \
            wp-config-sample.php  > wp-config.php
# Creating a local Git repository
$ git init
$ git add .
$ git commit -m 'initial project version'
```

4.7.4 Git deployment

When you create a new app, tsuru will display the Git remote that you should use. You can always get it using the command *app-info*:

```
$ tsuru app-info --app blog
Application: blog
Repository: git@192.168.50.4.nip.io:blog.git
Platform: php
Teams: admin
Address: blog.192.168.50.4.nip.io
Owner: admin@example.com
Team owner: admin
Deploys: 0
Pool: theonepool

App Plan:
+---------------+--------+------+-----------+--------+---------+
| Name          | Memory | Swap | Cpu Share | Router | Default |
+---------------+--------+------+-----------+--------+---------+
| autogenerated | 0 MB   | 0 MB | 100       |        | false   |
+---------------+--------+------+-----------+--------+---------+
```

The Git remote will be used to deploy your application using Git. You can just push to tsuru remote and your project will be deployed:

```
$ git push git@192.168.50.4.nip.io:blog.git master
Counting objects: 1295, done.
Delta compression using up to 4 threads.
Compressing objects: 100% (1271/1271), done.
Writing objects: 100% (1295/1295), 6.09 MiB | 5.65 MiB/s, done.
Total 1295 (delta 102), reused 0 (delta 0)
remote: text
remote: Deploying the PHP application...
remote: tar: Removing leading `/' from member names
########################################
#  OMIT DEPENDENCIES STEPS (see below)  #
########################################
remote:
remote: ---- Building application image ----
remote:  ---> Sending image to repository (51.40MB)
remote:  ---> Cleaning up
remote:
remote: ---- Starting 1 new unit ----
remote:  ---> Started unit 027c2a31a0...
remote:
remote: ---- Binding and checking 1 new units ----
remote:  ---> Bound and checked unit 027c2a31a0
remote:
remote: ---- Adding routes to 1 new units ----
remote:  ---> Added route to unit 027c2a31a0
remote:
remote: OK
To git@192.168.50.4.nip.io:blog.git
 * [new branch]      master -> master
```

If you get a "Permission denied (publickey).", make sure you're member of a team and have a public key added to tsuru. To add a key, use the command *key-add*:

```
$ tsuru key-add mykey ~/.ssh/id_dsa.pub
```

You can use `git remote add` to avoid typing the entire remote url every time you want to push:

```
$ git remote add tsuru git@192.168.50.4.nip.io:blog.git
```

Then you can run:

```
$ git push tsuru master
Everything up-to-date
```

And you will be also able to omit the `--app` flag from now on:

```
$ tsuru app-info
Application: blog
Repository: git@192.168.50.4.nip.io:blog.git
Platform: php
Teams: admin
Address: blog.192.168.50.4.nip.io
Owner: admin@example.com
Team owner: admin
Deploys: 1
Pool: theonepool
Units: 1
+------------+---------+
| Unit       | State   |
+------------+---------+
| 027c2a31a0 | started |
+------------+---------+

App Plan:
+---------------+--------+------+-----------+--------+---------+
| Name          | Memory | Swap | Cpu Share | Router | Default |
+---------------+--------+------+-----------+--------+---------+
| autogenerated | 0 MB   | 0 MB | 100       |        | false   |
+---------------+--------+------+-----------+--------+---------+
```

4.7.5 Listing dependencies

In the last section we omitted the dependencies step of deploy. In tsuru, an application can have two kinds of dependencies:

- **Operating system dependencies**, represented by packages in the package manager of the underlying operating system (e.g.: `yum` and `apt-get`);
- **Platform dependencies**, represented by packages in the package manager of the platform/language (e.g. in Python, `pip`).

All `apt-get` dependencies must be specified in a `requirements.apt` file, located in the root of your application, and pip dependencies must be located in a file called `requirements.txt`, also in the root of the application. Since we will use MySQL with PHP, we need to install the package depends on just one `apt-get` package: `php5-mysql`, so here is how `requirements.apt` looks like:

```
php5-mysql
```

You can see the complete output of installing these dependencies below:

```
% git push tsuru master
####################################
#                OMIT              #
####################################
Counting objects: 1155, done.
Delta compression using up to 4 threads.
Compressing objects: 100% (1124/1124), done.
Writing objects: 100% (1155/1155), 4.01 MiB | 327 KiB/s, done.
Total 1155 (delta 65), reused 0 (delta 0)
remote: Cloning into '/home/application/current'...
remote: Reading package lists...
remote: Building dependency tree...
remote: Reading state information...
remote: The following extra packages will be installed:
remote:   libmysqlclient18 mysql-common
remote: The following NEW packages will be installed:
remote:   libmysqlclient18 mysql-common php5-mysql
remote: 0 upgraded, 3 newly installed, 0 to remove and 0 not upgraded.
remote: Need to get 1042 kB of archives.
remote: After this operation, 3928 kB of additional disk space will be used.
remote: Get:1 http://archive.ubuntu.com/ubuntu/ quantal/main mysql-common all 5.5.27-0ubuntu2 [13.7 k
remote: Get:2 http://archive.ubuntu.com/ubuntu/ quantal/main libmysqlclient18 amd64 5.5.27-0ubuntu2
remote: Get:3 http://archive.ubuntu.com/ubuntu/ quantal/main php5-mysql amd64 5.4.6-1ubuntu1 [79.0 kB
remote: Fetched 1042 kB in 1s (739 kB/s)
remote: Selecting previously unselected package mysql-common.
remote: (Reading database ... 23874 files and directories currently installed.)
remote: Unpacking mysql-common (from .../mysql-common_5.5.27-0ubuntu2_all.deb) ...
remote: Selecting previously unselected package libmysqlclient18:amd64.
remote: Unpacking libmysqlclient18:amd64 (from .../libmysqlclient18_5.5.27-0ubuntu2_amd64.deb) ...
remote: Selecting previously unselected package php5-mysql.
remote: Unpacking php5-mysql (from .../php5-mysql_5.4.6-1ubuntu1_amd64.deb) ...
remote: Processing triggers for libapache2-mod-php5 ...
remote:  * Reloading web server config
remote:    ...done.
remote: Setting up mysql-common (5.5.27-0ubuntu2) ...
remote: Setting up libmysqlclient18:amd64 (5.5.27-0ubuntu2) ...
remote: Setting up php5-mysql (5.4.6-1ubuntu1) ...
remote: Processing triggers for libc-bin ...
remote: ldconfig deferred processing now taking place
remote: Processing triggers for libapache2-mod-php5 ...
remote:  * Reloading web server config
remote:    ...done.
remote: sudo: unable to resolve host 8cf20f4da877
remote: sudo: unable to resolve host 8cf20f4da877
remote: debconf: unable to initialize frontend: Dialog
remote: debconf: (Dialog frontend will not work on a dumb terminal, an emacs shell buffer, or without
remote: debconf: falling back to frontend: Readline
remote: debconf: unable to initialize frontend: Dialog
remote: debconf: (Dialog frontend will not work on a dumb terminal, an emacs shell buffer, or without
remote: debconf: falling back to frontend: Readline
remote:
remote: Creating config file /etc/php5/mods-available/mysql.ini with new version
remote: debconf: unable to initialize frontend: Dialog
remote: debconf: (Dialog frontend will not work on a dumb terminal, an emacs shell buffer, or without
remote: debconf: falling back to frontend: Readline
remote:
remote: Creating config file /etc/php5/mods-available/mysqli.ini with new version
remote: debconf: unable to initialize frontend: Dialog
```

```
remote: debconf: (Dialog frontend will not work on a dumb terminal, an emacs shell buffer, or without
remote: debconf: falling back to frontend: Readline
remote:
remote: Creating config file /etc/php5/mods-available/pdo_mysql.ini with new version
remote:
remote:  ---> App will be restarted, please check its log for more details...
remote:
To git@192.168.50.4.nip.io:blog.git
 * [new branch]      master -> master
```

4.7.6 Running the application

As you can see, in the deploy output there is a step described as "App will be restarted". In this step, tsuru will restart your app if it's running, or start it if it's not. Now that the app is deployed, you can access it from your browser, getting the IP or host listed in `app-list` and opening it. For example, in the list below:

```
$ tsuru app-list
+-------------+-------------------------+---------------------+
| Application | Units State Summary     | Address             |
+-------------+-------------------------+---------------------+
| blog        | 1 of 1 units in-service | blog.cloud.tsuru.io |
+-------------+-------------------------+---------------------+
```

4.7.7 Using services

Now that php is running, we can accesss the application in the browser, but we get a database connection error: *"Error establishing a database connection"*. This error means that we can't connect to MySQL. That's because we should not connect to MySQL on localhost, we must use a service. The service workflow can be resumed to two steps:

1. Create a service instance
2. Bind the service instance to the app

But how can I see what services are available? Easy! Use the command *service-list*:

```
$ tsuru service-list
+----------------+-----------+
| Services       | Instances |
+----------------+-----------+
| elastic-search |           |
| mysql          |           |
+----------------+-----------+
```

The output from `service-list` above says that there are two available services: "elastic-search" and "mysql", and no instances. To create our MySQL instance, we need to run the command *service-add*:

```
$ tsuru service-add mysql blogsql
Service successfully added.
```

Now, if we run `service-list` again, we will see our new service instance in the list:

```
$ tsuru service-list
+----------------+-----------+
| Services       | Instances |
+----------------+-----------+
| elastic-search |           |
```

```
| mysql          | blogsql    |
+----------------+------------+
```

To bind the service instance to the application, we use the command *service-bind*:

```
$ tsuru service-bind blogsql
Instance blogsql is now bound to the app blog.

The following environment variables are now available for use in your app:

- MYSQL_PORT
- MYSQL_PASSWORD
- MYSQL_USER
- MYSQL_HOST
- MYSQL_DATABASE_NAME

For more details, please check the documentation for the service, using service-doc command.
```

As you can see from bind output, we use environment variables to connect to the MySQL server. Next step would be update the `wp-config.php` to use these variables to connect in the database:

```
$ grep getenv wp-config.php
define('DB_NAME', getenv('MYSQL_DATABASE_NAME'));
define('DB_USER', getenv('MYSQL_USER'));
define('DB_PASSWORD', getenv('MYSQL_PASSWORD'));
define('DB_HOST', getenv('MYSQL_HOST'));
```

You can extend your wordpress installing plugins into your repository. In the example below, we are adding the Amazon S3 capability to wordpress, just installing 2 more plugins: Amazon S3 and Cloudfront + Amazon Web Services. It's the right way to store content files into tsuru.

```
$ cd wp-content/plugins/
$ wget http://downloads.wordpress.org/plugin/amazon-web-services.0.1.zip
$ wget http://downloads.wordpress.org/plugin/amazon-s3-and-cloudfront.0.6.1.zip
$ unzip amazon-web-services.0.1.zip
$ unzip amazon-s3-and-cloudfront.0.6.1.zip
$ rm -f amazon-web-services.0.1.zip amazon-s3-and-cloudfront.0.6.1.zip
$ git add amazon-web-services/ amazon-s3-and-cloudfront/
```

Now you need to add the amazon AWS_ACCESS_KEY_ID and AWS_SECRET_ACCESS_KEY environments support into wp-config.php. You could add these environments right after the WP_DEBUG as below:

```
$ grep -A2 define.*WP_DEBUG  wp-config.php
define('WP_DEBUG', false);
define('AWS_ACCESS_KEY_ID', getenv('AWS_ACCESS_KEY_ID'));
define('AWS_SECRET_ACCESS_KEY', getenv('AWS_SECRET_ACCESS_KEY'));
$ git add wp-config.php
$ git commit -m 'adding plugins for S3'
$ git push tsuru master
```

Now, just inject the right values for these environments with tsuru env-set as below:

```
$ tsuru env-set AWS_ACCESS_KEY_ID="xxx" AWS_SECRET_ACCESS_KEY="xxxxx" -a blog
```

It's done! Now we have a PHP project deployed on tsuru, with S3 support using a MySQL service.

4.7.8 Customizing the platform

The PHP platform supports customizations in the frontend and the interpreter, for more details, check the README of the platform.

4.7.9 Going further

For more information, you can dig into tsuru docs, or read complete instructions of use for the tsuru command.

4.8 Using Buildpacks

tsuru supports deploying applications via Heroku Buildpacks.

Buildpacks are useful if you're interested in following Heroku's best practices for building applications or if you are deploying an application that already runs on Heroku.

tsuru uses Buildstep Docker image to make deploy using buildpacks possible.

4.8.1 Creating an Application

What do you need is create an application using *buildpack* platform:

```
$ tsuru app-create myapp buildpack
```

4.8.2 Deploying your Application

Use *git push* to deploy your application.

```
$ git push <REMOTE-URL> master
```

4.8.3 Included Buildpacks

A number of buildpacks come bundled by default:

- https://github.com/heroku/heroku-buildpack-ruby.git
- https://github.com/heroku/heroku-buildpack-nodejs.git
- https://github.com/heroku/heroku-buildpack-java.git
- https://github.com/heroku/heroku-buildpack-play.git
- https://github.com/heroku/heroku-buildpack-python.git
- https://github.com/heroku/heroku-buildpack-scala.git
- https://github.com/heroku/heroku-buildpack-clojure.git
- https://github.com/heroku/heroku-buildpack-gradle.git
- https://github.com/heroku/heroku-buildpack-grails.git
- https://github.com/CHH/heroku-buildpack-php.git
- https://github.com/kr/heroku-buildpack-go.git

- https://github.com/oortcloud/heroku-buildpack-meteorite.git
- https://github.com/miyagawa/heroku-buildpack-perl.git
- https://github.com/igrigorik/heroku-buildpack-dart.git
- https://github.com/rhy-jot/buildpack-nginx.git
- https://github.com/Kloadut/heroku-buildpack-static-apache.git
- https://github.com/bacongobbler/heroku-buildpack-jekyll.git
- https://github.com/ddollar/heroku-buildpack-multi.git

tsuru will cycle through the bin/detect script of each buildpack to match the code you are pushing.

4.8.4 Using a Custom Buildpack

To use a custom buildpack, set the BUILDPACK_URL environment variable.

```
$ tsuru env-set BUILDPACK_URL=https://github.com/dpiddy/heroku-buildpack-ruby-minimal
```

On your next *git push*, the custom buildpack will be used.

4.8.5 Creating your own Buildpack

You can follow this Heroku documentation to learn how to create your own Buildpack: https://devcenter.heroku.com/articles/buildpack-api.

4.9 Recovering an application

Your application may be down for a number of reasons. This page can help you discover why and guide you to fix the problem.

4.9.1 Check your application logs

tsuru aggregates stdout and stderr from every application process making it easier to troubleshoot problems.

To know more how the tsuru log works see the log documentation.

4.9.2 Restart your application

Some application issues are solved by a simple restart. For example, your application may need to be restarted after a schema change to your database.

```
$ tsuru app-restart -a appname
```

4.9.3 Checking the status of application units

```
$ tsuru app-info -a appname
```

4.9.4 Open a shell to the application

You can also use *tsuru app-shell* to open a remote shell to one of the units of the application.

```
$ tsuru app-shell -a appname
```

You can also specify the unit ID to connect:

```
$ tsuru app-shell -a appname <container-id>
```

4.10 Logging

tsuru aggregates stdout and stderr from every application process making it easier to troubleshoot problems. To use the log make sure that your application is sending the log to stdout and stderr.

4.10.1 Watch your logs

To see the logs for your application. You can use the *tsuru app-log* command:

```
$ tsuru app-log -a <appname>
2014-12-11 16:36:17 -0200 [tsuru][api]:  ---> Removed route from unit 1d913e0910
2014-12-11 16:36:17 -0200 [tsuru][api]: ---- Removing 1 old unit ----
2014-12-11 16:36:22 -0200 [app][11f863b2c14b]: Starting gunicorn 18.0
2014-12-11 16:36:22 -0200 [app][11f863b2c14b]: Listening at: http://0.0.0.0:8100 (51)
2014-12-11 16:36:22 -0200 [app][11f863b2c14b]: Using worker: sync
2014-12-11 16:36:22 -0200 [app][11f863b2c14b]: Booting worker with pid: 60
2014-12-11 16:36:28 -0200 [tsuru][api]:  ---> Removed old unit 1/1
```

By default is showed the last ten log lines. If you want see more lines, you can use the *-l/–lines* parameter:

```
$ tsuru app-log -a <appname> --lines 100
```

Filtering

You can filter logs by unit and by source.

To filter by unit you should use *-u/–unit* parameter:

```
$ tsuru app-log -a <appname> --unit 11f863b2c14b
2014-12-11 16:36:22 -0200 [app][11f863b2c14b]: Starting gunicorn 18.0
2014-12-11 16:36:22 -0200 [app][11f863b2c14b]: Listening at: http://0.0.0.0:8100 (51)
2014-12-11 16:36:22 -0200 [app][11f863b2c14b]: Using worker: sync
```

See also:

To get the unit id you can use the *tsuru app-info -a <appname>* command.

The log can be sent by your process or by tsuru api. To filter by source you should use *-s/–source* parameter:

```
$ tsuru app-log -a <appname> --source app
2014-12-11 16:36:22 -0200 [app][11f863b2c14b]: Starting gunicorn 18.0
2014-12-11 16:36:22 -0200 [app][11f863b2c14b]: Listening at: http://0.0.0.0:8100 (51)
2014-12-11 16:36:22 -0200 [app][11f863b2c14b]: Using worker: sync

$ tsuru app-log -a <appname> --source tsuru
```

```
2014-12-11 16:36:17 -0200 [tsuru][api]:  ---> Removed route from unit 1d913e0910
2014-12-11 16:36:17 -0200 [tsuru][api]: ---- Removing 1 old unit ----
```

Realtime logging

tsuru app-log has a *-f/–follow* option that causes the log to not stop and wait for the new log data. With this option you can see in real time the behaviour of your application that is useful to debug problems:

```
$ tsuru app-log -a <appname> --follow
```

You can close the session pressing Ctrl-C.

Limitations

The tsuru native log system is designed to be fast and show the recent log of your application. The tsuru log doesn't store all log entries for your application.

If you want to store and see all log entries you should use an external log aggregator.

4.10.2 Using an external log aggregator

You can also send the log to an external log aggregator. To do this, tsuru uses the Syslog protocol.

To use Syslog you should set the following environment variables in your application:

```
TSURU_SYSLOG_SERVER
TSURU_SYSLOG_PORT (probably 514)
TSURU_SYSLOG_FACILITY (something like local0)
TSURU_SYSLOG_SOCKET (tcp or udp)
```

You can use the command *tsuru env-set* to set these enviroment variables in your application:

```
$ tsuru env-set -a myapp TSURU_SYSLOG_SERVER=myserver.com TSURU_SYSLOG_PORT=514 TSURU_SYSLOG_FACILITY
```

4.11 Procfile

Procfile is a simple text file called *Procfile* that describe the components required to run an applications. It is the way to tell to *tsuru* how to run your applications.

This document describes some of the more advances features of and the Procfile ecosystem.

A *Procfile* should look like:

```
web: gunicorn -w 3 wsgi
```

4.11.1 Syntax

Procfile is a plain text file called *Procfile* placed at the root of your application.

Each project should be represented by a name and a command, like below:

```
<name>: <command>
```

The *name* is a string which may contain alphanumerics and underscores and identifies one type of process.

command is a shell commandline which will be executed to spawn a process.

4.11.2 Environment variables

You can reference yours environment variables in the command:

```
web: ./manage.py runserver 0.0.0.0:$PORT
```

For more information about *Procfile* you can see the honcho documentation about *Procfiles*: http://honcho.rtfd.org/en/latest/using_procfiles.html.

4.12 tsuru.yaml

tsuru.yaml is a special file located in the root of the application. The name of the file may be `tsuru.yaml` or `tsuru.yml`.

This file is used to describe certain aspects of your app. Currently it describes information about deployment hooks and deployment time health checks. How to use this features is described below.

4.12.1 Deployment hooks

tsuru provides some deployment hooks, like `restart:before`, `restart:after` and `build`. Deployment hooks allow developers to run commands before and after some commands.

Here is an example about how to declare this hooks in your tsuru.yaml file:

```
hooks:
  restart:
    before:
      - python manage.py generate_local_file
    after:
      - python manage.py clear_local_cache
  build:
    - python manage.py collectstatic --noinput
    - python manage.py compress
```

tsuru supports the following hooks:

- `restart:before`: this hook lists commands that will run before the unit is restarted. Commands listed in this hook will run once per unit. For instance, imagine there's an app with two units and the `tsuru.yaml` file listed above. The command **python manage.py generate_local_file** would run two times, once per unit.
- `restart:after`: this hook is like before-each, but runs after restarting a unit.
- `build`: this hook lists commands that will be run during deploy, when the image is being generated.

4.12.2 Healthcheck

You can declare a health check in your tsuru.yaml file. This health check will be called during the deployment process and tsuru will make sure this health check is passing before continuing with the deployment process.

If tsuru fails to run the health check successfully it will abort the deployment before switching the router to point to the new units, so your application will never be unresponsive. You can configure the maximum time to wait for the application to respond with the `docker:healthcheck:max-time` config.

Here is how you can configure a health check in your yaml file:

```
healthcheck:
  path: /healthcheck
  method: GET
  status: 200
  match: .*OKAY.*
  allowed_failures: 0
```

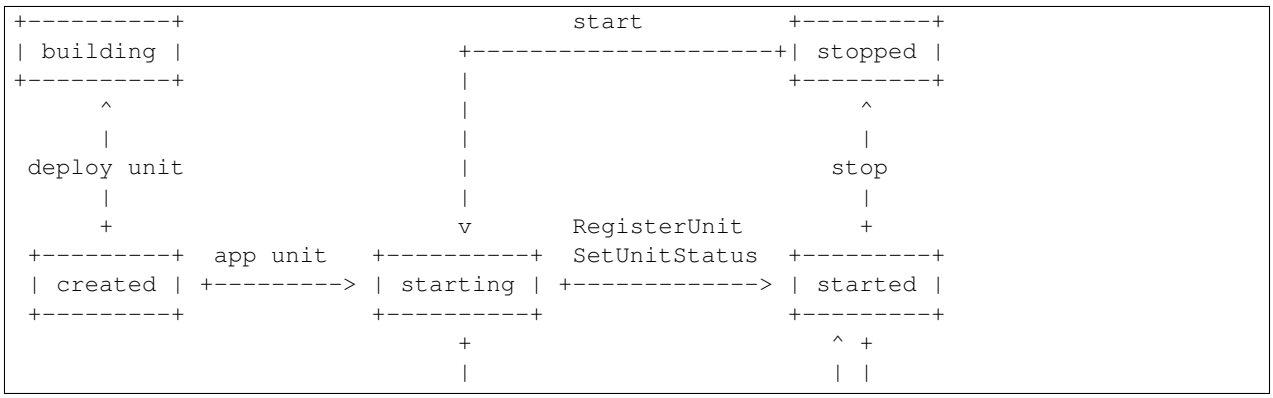
- `healthcheck:path`: Which path to call in your application. This path will be called for each unit. It is the only mandatory field, if it's not set your health check will be ignored.
- `healthcheck:method`: The method used to make the http request. Defaults to GET.
- `healthcheck:status`: The expected response code for the request. Defaults to 200.
- `healthcheck:match`: A regular expression to be matched against the request body. If it's not set the body won't be read and only the status code will be checked. This regular expression uses Go syntax and runs with `.` matching `\n` (`s` flag).
- `healthcheck:allowed_failures`: The number of allowed failures before that the health check consider the application as unhealthy. Defaults to 0.

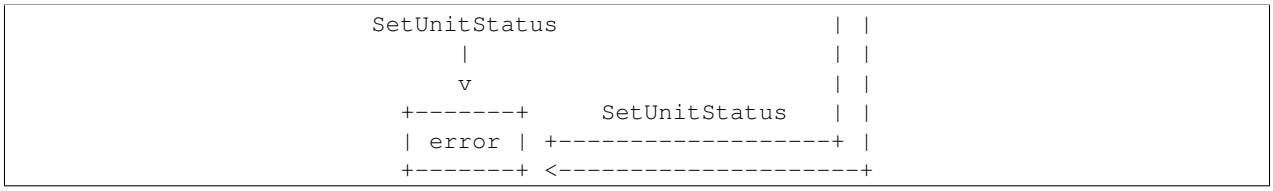
4.13 Unit states

The unit status is the way to know what is happening with a unit. You can use the *tsuru app-info -a <appname>* to see the unit status:

```
$ tsuru app-info -a tsuru-dashboard
Application: tsuru-dashboard
Repository: git@localhost:tsuru-dashboard.git
Platform: python
...
Units: 1
+------------+---------+
| Unit       | State   |
+------------+---------+
| 9cf863c2c1 | started |
+------------+---------+
```

The unit state flow is:





- *created*: is the initial status of an unit.
- *building*: is the status for units being provisioned by the provisioner, like during deployment.
- *error*: is the status for units that failed to start, because of an application error.
- *starting*: is set when the container is started in docker.
- *started*: is for cases where the unit is up and running.
- *stopped*: is for cases where the unit has been stopped.

4.14 tsuru client plugins

4.14.1 Installing a plugin

Let's install a plugin. There are two ways to install a plugin. The first way is to move your plugin to `$HOME/.tsuru/plugins`. The other way is to use the command `tsuru plugin-install`.

`tsuru plugin-install` will download the plugin file to `$HOME/.tsuru/plugins`. The syntax for this command is:

```
$ tsuru plugin-install <plugin-name> <plugin-url>
```

4.14.2 Listing installed plugins

To list all installed plugins, users can use the command `tsuru plugin-list`:

```
$ tsuru plugin-list
plugin1
plugin2
```

4.14.3 Executing a plugin

To execute a plugin just follow this pattern `tsuru <plugin-name> <args>`:

```
$ tsuru <plugin-name>
<plugin-output>
```

4.14.4 Removing a plugin

To remove a plugin just use the command `tsuru plugin-remove` passing the name of the plugin as argument:

```
$ tsuru plugin-remove <plugin-name>
Plugin "<plugin-name>" successfully removed!
```

4.14.5 Creating your own plugin

All you need to do is to create a new file that can be executed. You can use Shell Script, Python, Ruby, etc.

As an example, we're going to show how to create a Hello world plugin, that just prints "hello world!" in the screen. Let's use Shell Script in this plugin:

```
#!/bin/bash -e
echo "hello world!"
```

You can use the gist (https://gist.github.com) as host for your plugin, and run `tsuru plugin-install` to install it:

```
$ tsuru plugin-install hello https://gist.githubusercontent.com/fsouza/702a767f48b0ceaafebe/raw/9bcdf
```

4.15 Application Deployment

This document provides a high-level description on how application deployment works on tsuru.

4.15.1 Preparing Your Application

If you follow the 12 Factor app principles you shouldn't have to change your application in order to deploy it on tsuru. Here is what an application need to go on a tsuru cloud:

1. Well defined requirements, both, on language level and operational system level
2. Configuration of external resources using environment variables
3. A Procfile to tell how your process should be run

Let's go a little deeper through each of those topics.

1. Requirements

Every well writen application nowdays has well defined dependencies. In Python, everything is on a requirements.txt or like file, in Ruby, they go on Gemfile, Node.js has the package.json, and so on. Some of those dependencies also have operational system level dependencies, like the Nokogiri Ruby gem or MySQL-Python package, tsuru bootstraps units as clean as possible, so you also have to declare those operational system requirements you need on a file called requirements.apt. This files should have the packages declared one per-line and look like that:

```
python-dev
libmysqlclient-dev
```

2. Configuration With Environment Variables

Everything that vary between deploys (on different environments, like development or production) should be managed by environment variables. tsuru takes this principle very seriously, so all services available for usage in tsuru that requires some sort of configuration does it via environment variables so you have no pain while deploying on different environments using tsuru.

For instance, if you are going to use a database service on tsuru, like MySQL, when you bind your application into the service, tsuru will receive from the service API everything you need to connect with MySQL, e.g: user name, password, url and database name. Having this information, tsuru will export on every unit your application has the

equivalent environment variables with their values. The names of those variables are defined by the service providing them, in this case, the MySQL service.

Let's take a look at the settings of tsuru hosted application built with Django:

```
import os

DATABASES = {
    "default": {
        "ENGINE": "django.db.backends.mysql",
        "NAME": os.environ.get("MYSQLAPI_DB_NAME"),
        "USER": os.environ.get("MYSQLAPI_DB_USER"),
        "PASSWORD": os.environ.get("MYSQLAPI_DB_PASSWORD"),
        "HOST": os.environ.get("MYSQLAPI_HOST"),
        "PORT": "",
    }
}
```

You might be asking yourself "How am I going to know those variables names?", but don't fear! When you bind your application with tsuru, it'll return all variables the service asked tsuru to export on your application's units (without the values, since you are not gonna need them), if you lost the environments on your terminal history, again, don't fear! You can always check which service made what variables available to your application using the <insert command here>.

4.16 Choose a pool to deploy your app

tsuru has a concept of pool, a group of machines that will run the application code. Pools are defined by the cloud admin as needed and users can choose one of them in the moment of app creation.

Users can see which pools are available using the command *tsuru pool-list*:

```
$ tsuru pool-list

+---------+--------------+
| Team    | Pools        |
+---------+--------------+
| team1   | pool1, pool2 |
+---------+--------------+
```

So, in *app-create*, users can choose the pool using the *-o/–pool pool_name* flag:

```
$ tsuru app-create app_name platform -o pool1
```

There's no need to specify the pool when the user has access to only one pool.

CHAPTER 5

Services

5.1 API workflow

tsuru sends requests to the service API to the following actions:

- create a new instance of the service (`tsuru service-add`)
- bind an app with the service instance (`tsuru service-bind`)
- unbind an app from the service instance (`tsuru service-unbind`)
- destroy the service instance (`tsuru service-remove`)
- check the status of the service instance (`tsuru service-status`)
- display additional info about a service, including instances and available plans (`tsuru service-info`)

5.1.1 Authentication

tsuru will authenticate with the service API using HTTP basic authentication. The user can be username or name of the service, and the password is defined in the *service manifest*.

5.1.2 Content-types

tsuru uses `application/x-www-form-urlencoded` in requests and expect `application/json` in responses.

Here is an example of a request from tsuru, to the service API:

```
POST /resources HTTP/1.1
Host: myserviceapi.com
User-Agent: Go 1.1 package http
Content-Length: 38
Accept: application/json
Authorization: Basic dXNlcjpwYXNzd29yZA==
Content-Type: application/x-www-form-urlencoded

name=myinstance&plan=small&team=myteam
```

5.1.3 Listing available plans

tsuru will list the available plans whenever the user issues the command `service-info`

```
$ tsuru service-info mysql
```

It will display all instances of the service that the user has access to, and also the list of plans, that tsuru gets from the service API by issuing a GET on `/resources/plans`. Example of request:

```
GET /resources/plans HTTP/1.1
Host: myserviceapi.com
User-Agent: Go 1.1 package http
Accept: application/json
Authorization: Basic dXNlcjpwYXNzd29yZA==
Content-Type: application/x-www-form-urlencoded
```

The API should return the following HTTP response codes with the respective response body:

- 200: if the operation has succeeded. The response body should include the list of the plans, in JSON format. Each plan contains a "name" and a "description". Example of response:

```
HTTP/1.1 200 OK
Content-Type: application/json; charset=UTF-8

[{"name":"small","description":"plan for small instances"},
 {"name":"medium","description":"plan for medium instances"},
 {"name":"huge","description":"plan for huge instances"}]
```

In case of failure, the service API should return the status 500, explaining what happened in the response body.

5.1.4 Creating a new instance

This process begins when a tsuru customer creates an instance of the service via command line tool:

```
$ tsuru service-add mysql mysql_instance
```

tsuru calls the service API to create a new instance via POST on `/resources` (please notice that tsuru does not include a trailing slash) with the name, plan and the team that owns the instance. Example of request:

```
POST /resources HTTP/1.1
Host: myserviceapi.com
Content-Length: 56
User-Agent: Go 1.1 package http
Accept: application/json
Authorization: Basic dXNlcjpwYXNzd29yZA==
Content-Type: application/x-www-form-urlencoded

name=mysql_instance&plan=small&team=myteam&user=username
```

The API should return the following HTTP response codes with the respective response body:

- 201: when the instance is successfully created. There's no need to include any body, as tsuru doesn't expect to get any content back in case of success.
- 500: in case of any failure in the operation. tsuru expects that the service API includes an explanation of the failure in the response body.

5.1.5 Binding an app to a service instance

This process begins when a tsuru customer binds an app to an instance of the service via command line tool:

```
$ tsuru service-bind mysql_instance --app my_app
```

Now, tsuru services has two bind endpoints: `/resources/<service-instance-name>/bind` and `/resources/<service-instance-name>/bind-app`. The first endpoint will be called every time an app adds an unit. This endpoint is a POST with app-host and unit-host, where app-host represents the host to which the app is accessible, and unit-host is the address of the unit. Example of request:

```
POST /resources/myinstance/bind HTTP/1.1
Host: myserviceapi.com
User-Agent: Go 1.1 package http
Content-Length: 48
Accept: application/json
Authorization: Basic dXNlcjpwYXNzd29yZA==
Content-Type: application/x-www-form-urlencoded

app-host=myapp.cloud.tsuru.io&unit-host=10.4.3.2
```

The second endpoint `/resources/<service-instance-name>/bind-app` will be called once when an app is bound to a service. This endpoint is a POST with app-host, where app-host represents the host to which the app is accessible. Example of request:

```
POST /resources/myinstance/bind-app HTTP/1.1
Host: myserviceapi.com
User-Agent: Go 1.1 package http
Content-Length: 48
Accept: application/json
Authorization: Basic dXNlcjpwYXNzd29yZA==
Content-Type: application/x-www-form-urlencoded

app-host=myapp.cloud.tsuru.io
```

The service API should return the following HTTP response code with the respective response body:

- 201: if the app has been successfully bound to the instance. The response body must be a JSON containing the environment variables from this instance that should be exported in the app in order to connect to the instance. If the service does not export any environment variable, it can return `null` or `{}` in the response body. Example of response:

```
HTTP/1.1 201 CREATED
Content-Type: application/json; charset=UTF-8

{"MYSQL_HOST":"10.10.10.10","MYSQL_PORT":3306,
 "MYSQL_USER":"ROOT","MYSQL_PASSWORD":"s3cr3t",
 "MYSQL_DATABASE_NAME":"myapp"}
```

Status codes for errors in the process:

- 404: if the service instance does not exist. There's no need to include anything in the response body.
- 412: if the service instance is still being provisioned, and not ready for binding yet. The service API may include an explanation of the failure in the response body.
- 500: in case of any failure in the operation. tsuru expects that the service API includes an explanation of the failure in the response body.

5.1.6 Unbind an app from a service instance

This process begins when a tsuru customer unbinds an app from an instance of the service via command line:

```
$ tsuru service-unbind mysql_instance --app my_app
```

Now, tsuru services has two unbind endpoints: `/resources/<service-instance-name>/bind` and `/resources/<service-instance-name>/bind-app`. The first endpoint will be called every time an app removes an unit. This endpoint is a DELETE with app-host and unit-host. Example of request:

```
DELETE /resources/myinstance/bind HTTP/1.1
Host: myserviceapi.com
User-Agent: Go 1.1 package http
Accept: application/json
Authorization: Basic dXNlcjpwYXNzd29yZA==
Content-Type: application/x-www-form-urlencoded

app-host=myapp.cloud.tsuru.io&unit-host=10.4.3.2
```

The second endpoint `/resources/<service-instance-name>/bind-app` will be called once when the binding between a service and an application is removed. This endpoint is a DELETE with app-host. Example of request:

```
DELETE /resources/myinstance/bind-app HTTP/1.1
Host: myserviceapi.com
User-Agent: Go 1.1 package http
Accept: application/json
Authorization: Basic dXNlcjpwYXNzd29yZA==
Content-Type: application/x-www-form-urlencoded

app-host=myapp.cloud.tsuru.io
```

The API should return the following HTTP response code with the respective response body:

- 200: if the operation has succeed and the app is not bound to the service instance anymore. There's no need to include anything in the response body.
- 404: if the service instance does not exist. There's no need to include anything in the response body.
- 500: in case of any failure in the operation. tsuru expects that the service API includes an explanation of the failure in the response body.

5.1.7 Removing an instance

This process begins when a tsuru customer removes an instance of the service via command line:

```
$ tsuru service-remove mysql_instance -y
```

tsuru calls the service API to remove the instancevia DELETE on `/resources/<service-name>` (please notice that tsuru does not include a trailing slash). Example of request:

```
DELETE /resources/myinstance HTTP/1.1
Host: myserviceapi.com
User-Agent: Go 1.1 package http
Accept: application/json
Authorization: Basic dXNlcjpwYXNzd29yZA==
Content-Type: application/x-www-form-urlencoded
```

The API should return the following HTTP response codes with the respective response body:

- 200: if the service instance has been successfully removed. There's no need to include anything in the response body.
- 404: if the service instance does not exist. There's no need to include anything in the response body.
- 500: in case of any failure in the operation. tsuru expects that the service API includes an explanation of the failure in the response body.

5.1.8 Checking the status of an instance

This process begins when a tsuru customer wants to check the status of an instance via command line:

```
$ tsuru service-status mysql_instance
```

tsuru calls the service API to check the status of the instance via GET on `/resources/mysql_instance/status` (please notice that tsuru does not include a trailing slash). Example of request:

```
GET /resources/myinstance/status HTTP/1.1
Host: myserviceapi.com
User-Agent: Go 1.1 package http
Accept: application/json
Authorization: Basic dXNlcjpwYXNzd29yZA==
Content-Type: application/x-www-form-urlencoded
```

The API should return the following HTTP response code, with the respective response body:

- 202: the instance is still being provisioned (pending). There's no need to include anything in the response body.
- 204: the instance is running and ready for connections (running).
- 500: the instance is not running, nor ready for connections. tsuru expects an explanation of what happened in the response body.

5.1.9 Additional info about an instance

When the user run `tsuru service-info <service>`, tsuru will get informations from all instances. This is an optional endpoint in the service API. Some services does not provide any extra information for instances. Example of request:

```
GET /resources/myinstance HTTP/1.1
Host: myserviceapi.com
User-Agent: Go 1.1 package http
Accept: application/json
Authorization: Basic dXNlcjpwYXNzd29yZA==
Content-Type: application/x-www-form-urlencoded
```

The API should return the following HTTP response codes:

- 404: when the API doesn't have extra info about the service instance. There's no need to include anything in the response body.
- 200: when there's extra information of the service instance. The response body must be a JSON containing a list of items. Each item is a JSON object combosed by a label and a value. Example response:

```
HTTP/1.1 200 OK
Content-Type: application/json; charset=UTF-8
```

```
[{"label":"my label","value":"my value"},
 {"label":"myLabel2.0","value":"my value 2.0"}]
```

5.2 Building your service

5.2.1 Overview

This document is a hands-on guide to turning your existing cloud service into a tsuru service.

In order to create a service you need to implement a provisioning API for your service, which tsuru will call using HTTP protocol when a customer creates a new instance or binds a service instance with an app.

You will also need to create a YAML document that will serve as the service manifest. We provide a command-line tool to help you to create this manifest and manage your service.

5.2.2 Creating your service API

To create your service API, you can use any programming language or framework. In this tutorial we will use Flask.

5.2.3 Authentication

tsuru uses basic authentication for authenticating the services, for more details, check the *service API workflow*.

Using Flask, you can manage basic authentication using a decorator described in this Flask snippet: http://flask.pocoo.org/snippets/8/.

Prerequisites

First, let's ensure that Python and pip are already installed:

```
$ python --version
Python 2.7.2

$ pip
Usage: pip COMMAND [OPTIONS]

pip: error: You must give a command (use "pip help" to see a list of commands)
```

For more information about how to install python you can see the Python download documentation and about how to install pip you can see the pip installation instructions.

Now, with python and pip installed, you can use pip to install Flask:

```
$ pip install flask
```

Now that Flask is installed, it's time to create a file called api.py and add the code needed to create a minimal Flask application:

```
from flask import Flask
app = Flask(__name__)

@app.route("/")
def hello():
```

```
    return "Hello World!"

if __name__ == "__main__":
    app.run()
```

For run this app you can do:

```
$ python api.py
 * Running on http://127.0.0.1:5000/
```

If you open your web browser and access the url http://127.0.0.1:5000/ you will see the message "Hello World!".

Then, you need to implement the resources of a tsuru service API, as described in the tsuru service API workflow.

Listing available plans

tsuru will get the list of available plans by issuing a GET request in the `/resources/plans` URL. Let's create the view that will handle this kind of request:

```
import json


@app.route("/resources/plans", methods=["GET"])
def plans():
    plans = [{"name": "small", "description": "small instance"},
             {"name": "medium", "description": "medium instance"},
             {"name": "big", "description": "big instance"},
             {"name": "giant", "description": "giant instance"}]
    return json.dumps(plans)
```

Creating new instances

For new instances tsuru sends a POST to /resources with the parameters needed for creating an instance. If the service instance is successfully created, your API should return 201 in status code.

Let's create the view for this action:

```
from flask import request


@app.route("/resources", methods=["POST"])
def add_instance():
    name = request.form.get("name")
    plan = request.form.get("plan")
    team = request.form.get("team")
    # use the given parameters to create the instance
    return "", 201
```

Binding instances to apps

In the bind action, tsuru calls your service via POST on `/resources/<service-instance-name>/bind-app` with the parameters needed for binding an app into a service instance.

If the bind operation succeeds, the API should return 201 as status code with the variables to be exported in the app environment on body in JSON format.

As an example, let's create a view that returns a json with a fake variable called "SOMEVAR" to be injected in the app environment:

```
import json

from flask import request


@app.route("/resources/<name>/bind-app", methods=["POST"])
def bind_app(name):
    app_host = request.form.get("app-host")
    # use name and app_host to bind the service instance and the #
    application
    envs = {"SOMEVAR": "somevalue"}
    return json.dumps(envs), 201
```

Unbinding instances from apps

In the unbind action, tsuru issues a `DELETE` request to the URL `/resources/<service-instance-name>/bind-app`.

If the unbind operation succeeds, the API should return 200 as status code. Let's create the view for this action:

```
@app.route("/resources/<name>/bind-app", methods=["DELETE"])
def unbind_app(name):
    app_host = request.form.get("app-host")
    # use name and app-host to remove the bind
    return "", 200
```

Whitelisting units

When binding and unbindin application and service instances, tsuru will also provide information about units that will have access to the service instance, so the service API can handle any required whitelisting (writing ACL rules to a network switch or authorizing access in a firewall, for example).

tsuru will send POST and DELETE requests to the route `/resources/<name>/bind`, with the host of the app and the unit, so any access control can be handled by the API:

```
@app.route("/resources/<name>/bind", methods=["POST", "DELETE"])
def access_control(name):
    app_host = request.form.get("app-host")
    unit_host = request.form.get("unit-host")
    # use unit-host and app-host, according to the access control tool, and
    # the request method.
    return "", 201
```

Removing instances

In the remove action, tsuru issues a DELETE request to the URL `/resources/<service_name>`.

If the service instance is successfully removed, the API should return 200 as status code.

Let's create a view for this action:

```
@app.route("/resources/<name>", methods=["DELETE"])
def remove_instance(name):
    # remove the instance named "name"
    return "", 200
```

Checking the status of an instance

To check the status of an instance, tsuru issues a GET request to the URL `/resources/<service_name>/status`. If the instance is ok, this URL should return 204.

Let's create a view for this action:

```
@app.route("/resources/<name>/status", methods=["GET"])
def status(name):
    # check the status of the instance named "name"
    return "", 204
```

The final code for our "fake API" developed in Flask is:

```
import json

from flask import Flask, request

app = Flask(__name__)


@app.route("/resources/plans", methods=["GET"])
def plans():
    plans = [{"name": "small", "description": "small instance"},
             {"name": "medium", "description": "medium instance"},
             {"name": "big", "description": "big instance"},
             {"name": "giant", "description": "giant instance"}]
    return json.dumps(plans)


@app.route("/resources", methods=["POST"])
def add_instance():
    name = request.form.get("name")
    plan = request.form.get("plan")
    team = request.form.get("team")
    # use the given parameters to create the instance
    return "", 201


@app.route("/resources/<name>/bind-app", methods=["POST"])
def bind_app(name):
    app_host = request.form.get("app-host")
    # use name and app_host to bind the service instance and the #
    application
    envs = {"SOMEVAR": "somevalue"}
    return json.dumps(envs), 201


@app.route("/resources/<name>/bind-app", methods=["DELETE"])
def unbind_app(name):
    app_host = request.form.get("app-host")
    # use name and app-host to remove the bind
    return "", 200


@app.route("/resources/<name>", methods=["DELETE"])
def remove_instance(name):
    # remove the instance named "name"
    return "", 200
```

```
@app.route("/resources/<name>/bind", methods=["POST", "DELETE"])
def access_control(name):
    app_host = request.form.get("app-host")
    unit_host = request.form.get("unit-host")
    # use unit-host and app-host, according to the access control tool, and
    # the request method.
    return "", 201


@app.route("/resources/<name>/status", methods=["GET"])
def status(name):
    # check the status of the instance named "name"
    return "", 204

if __name__ == "__main__":
    app.run()
```

5.2.4 Creating a service manifest

Using crane you can create a manifest template:

```
$ crane template
```

This will create a manifest.yaml in your current path with this content:

```
id: servicename
password: abc123
endpoint:
    production: production-endpoint.com
```

The manifest.yaml is used by crane to defined the ID, the password and the production endpoint of your service.

Change these information in the created manifest, and the *submit your service*:

```
id: servicename
username: username_to_auth
password: 1CWpoX2Zr46Jhc7u
endpoint:
  production: production-endpoint.com
    test: test-endpoint.com:8080
```

submit your service: *Submiting your service API*

5.2.5 Submiting your service API

To submit your service, you can run:

```
$ crane create manifest.yaml
```

For more details, check the service API workflow and the crane usage guide.

5.3 TSURU_SERVICES environment variable

tsuru exports an special environment variable in applications that use services, this variable is named `TSURU_SERVICES`. The value of this example is a JSON describing all services instances that the application uses. Here is an example of the value of this variable:

```
{
    "mysql": [
      {"instance_name": "mydb",
       "envs": {"DATABASE_NAME": "mydb",
                "DATABASE_USER": "mydb",
                "DATABASE_PASSWORD": "secret",
                "DATABASE_HOST": "mysql.mycompany.com"}
      },
      {"instance_name": "otherdb",
       "envs": {"DATABASE_NAME": "otherdb",
                "DATABASE_USER": "otherdb",
                "DATABASE_PASSWORD": "secret",
                "DATABASE_HOST": "mysql.mycompany.com"}
      }],
    "redis": [
      {"instance_name": "powerredis",
       "envs": {"REDIS_HOST": "remote.redis.company.com:6379"}
      }],
    "mongodb": []
}
```

As described in the structure, the value of the environment variable is a JSON object, where each key represents a service. In the example above, there are three services: mysql, redis and mongodb. Each service contains a list of service instances, and each instance have a name and a map of environment variables.

5.4 crane usage

crane is a command line for service providers/administrators on tsuru.

See the crane documentation for a full reference: https://tsuru-crane.readthedocs.org.

CHAPTER 6

Advanced topics

6.1 Metrics

Note: Currently tsuru supports statsd and graphite.

tsuru sends metrics data using statsd protocol and **tsuru-dashboard** (web interface) shows these data using graphite protocol.

6.1.1 Sending metrics

By default **tsuru** sends the metrics to *localhost:8125* on each unit. You can configure the statsd host and port defining the *STATSD_PORT* and *STATSD_HOST* environment variables.

Note: If you don't want to have your own statsd/graphite infrastructure, you can install a client to get the data from localhost and send to a private server that supports statsd protocol.

6.1.2 Metrics graph on tsuru-dashboard

tsuru-dashboard displays a graphic for each metric. To know where to get the metric data, the dashboards get the *GRAPHITE_HOST* environment variable from the application.

6.1.3 Kind of metrics

- net.connections - the number of connections established
- cpu_max - cpu utilization
- mem_max - memory utilization

6.2 Node Auto Scaling

Node auto scaling can be enabled by setting *docker:auto-scale:enabled* to true. It will try to add, remove and rebalance docker nodes used by tsuru.

Node scaling algorithms run in clusters of docker nodes, to specify how clusters will be formed you must tell tsuru how they should be grouped. This is done by setting *docker:auto-scale:group-by-metadata* configuration entry to the name of a metadata present in your nodes.

There are two different scaling algorithms that will be used, depending on how tsuru is configured: count based scaling, and memory based scaling.

6.2.1 Count based scaling

It's chosen if *docker:auto-scale:max-container-count* is set to a value > 0 in your tsuru configuration.

Adding nodes

Having *max-container-count* value as max, the number of nodes in cluster as $nodes$, and the total number of containers in all cluster's nodes as $total$, we get the number of free slots $free$ with:

$$free = max * nodes - total$$

If $free < 0$ then a new node will be added and tsuru will rebalance containers using the new node.

Removing nodes

Having *docker:auto-scale:scale-down-ratio* value $ratio$. tsuru will try to remove an existing node if:

$$free > max * ratio$$

Before removing a node tsuru will move it's containers to other nodes available in the cluster.

To avoid entering loops, removing and adding node, tsuru will require $ratio > 1$, if this is not true scaling will not run.

6.2.2 Memory based scaling

It's chosen if *docker:auto-scale:max-container-count* is not set and your scheduler is configured to use node's memory information, by setting *docker:scheduler:total-memory-metadata* and *docker:scheduler:max-used-memory*.

Adding nodes

Having the amount of memory necessary by the plan with the largest memory requirement as $maxPlanMemory$. A new node will be added if for all nodes the amount of unreserved memory ($unreserved$) satisfies:

$$unreserved < maxPlanMemory$$

Removing nodes

Considering the amount of memory necessary by the plan with the largest memory requirement as $maxPlanMemory$ and *docker:auto-scale:scale-down-ratio* value as $ratio$. A node will be removed if its current containers can be distributed across other nodes in the same pool and at least one node still has unreserved memory ($unreserved$) satisfying:

$$unreserved > maxPlanMemory * ratio$$

6.2.3 Rebalancing nodes

Rebalancing containers will be triggered when a new node is added or if rebalancing would decrease the difference of containers in nodes by a number greater than 2, regardless the scaling algorithm.

Also, rebalancing will not run if *docker:auto-scale:prevent-rebalance* is set to true.

6.2.4 Auto scale events

Each time tsuru tries to run an auto scale action (add, remove, or rebalance). It will create an auto scale event. This event will record the result of the auto scale action and possible errors that occurred during its execution.

You can list auto scale events with *tsuru-admin docker-autoscale-list*

6.2.5 Running auto scale once

Even if you have *docker:auto-scale:enabled* set to false, you can make tsuru trigger the execution of the auto scale algorithm by running *tsuru-admin docker- autoscale-run*.

CHAPTER 7

Contributing

- Source hosted at GitHub
- Report issues on GitHub Issues

Pull requests are very welcome! Make sure your patches are well tested and documented :)

7.1 Development environment

7.1.1 Coding style

Please follow these coding standards when writing code for inclusion in tsuru.

Formatting

- Follow the Go formatting style

Naming standards

New<Something>

is used the constructor of *Something*:

```
NewApp(name string) (*App, error)
```

Add<Something>

is a *method* of a type that has a collection of *Something*'s. *Should receive an instance of 'Something*:

```
func (a *App) AddUnit(u *Unit) error
```

Add

is a method of a collection that adds one or more elements:

```
func (a *AppList) Add(apps ...*App) error
```

Create<Something>

is a function that saves an instance of *Something*. Unlike `NewSomething`, the create function would create a persistent version of *Someting*. Storing it in the database, a remote API, the filesystem or wherever *Something* would be stored "forever".

Comes in two versions:

1. One that receives the instance of *Something* and returns an error:

```
        func CreateApp(a *App) error
```

2. Other that receives the required parameters and returns the an instance of *Something* and an error:

```
        func CreateUser(email string) (*User, error)
```

Delete<Something>

is a function that destroy an instance of *Something*. Destroying may involve processes like removing it from the database and some directory in the filesystem.

For example:

```
func DeleteApp(app *App) error
```

Would delete an application from the database, delete the repository, remove the entry in the router, and anything else that depends on the application.

It's also valid to write the function so it receives some other kind of values that is able to identify the instance of *Something*:

```
func DeleteApp(name string) error
```

Remove<Something>

is the opposite of Add<Something>.

Including the package in the name of the function

For functions, it's also possible to omit *Something* when the name of the package represents *Something*. For example, if there's a package named "app", the function CreateApp could be just "Create". The same applies to other functions. This way callers won't need to write verbose code like `something.CreateSomething`, preferring `something.Create`.

7.1.2 Building a development environment with Vagrant

First, make sure that one of the supported Vagrant providers, Vagrant, and Git are installed on your machine.

Then clone the tsuru-bootstrap project from GitHub:

```
$ git clone https://github.com/tsuru/tsuru-bootstrap.git
```

Enter the `tsuru-bootstrap` directory and execute `vagrant up`, defining the environment variable TSURU_NOW_OPTIONS as "–tsuru-from-source". It will take some time:

```
$ cd tsuru-bootstrap
$ TSURU_NOW_OPTIONS="--tsuru-from-source" vagrant up
```

You can optionally specify a provider with the `--provider` parameter. The following providers are configured in the Vagrantfile:

- VirtualBox
- EC2
- Parallels Desktop

Then configure the tsuru target with the address of the server that vagrant is using:

```
$ tsuru target-add development http://192.168.50.4:8080 -s
```

Now you can create your user and deploy your apps.

See this guide to to setup a development environment using Vagrant.

And follow our coding style guide.

7.2 Running the tests

You can use *make* to install all tsuru dependencies and run tests. It will also check if everything is ok with your *GOPATH* setup:

```
$ make
```

Please ensure that MongoDB and Redis are started before running the test suite. If you see some test failures with messages like "dial tcp 127.0.0.1:6379: connection refused" and "no reachable server", the most likely reason is that these services are not running.

If you just want to run the tests you can use *make test*.

```
$ make test
```

7.3 Writing docs

tsuru documentation is written using Sphinx, which uses RST. Check out these tools documentation to learn how to write and update the documentation for tsuru.

7.4 Building docs

In order to build the HTML docs, just run in a terminal window:

```
$ make doc
```

7.5 Community

7.5.1 irc channel

#tsuru channel on irc.freenode.net - chat with other tsuru users and developers.

7.5.2 Gitter

We're also on Gitter, check it out: https://gitter.im/tsuru/tsuru.

7.6 Release Process

tsuru major releases are guided by GitHub milestones. New releases should be generated by *make release version=new-version-number*.

CHAPTER 8

Reference

8.1 tsuru client usage

tsuru-client is the command line utility used by application developers, that will allow users to create, list, bind and manage apps.

See the tsuru-client documentation for a full reference: https://tsuru-client.readthedocs.org.

8.2 tsuru-admin usage

tsuru-admin command supports administrative operations on a tsuru server.

See the tsuru-admin documentation for a full reference: http://tsuru-admin.readthedocs.org.

8.3 crane usage

crane is a command line for service providers/administrators on tsuru.

See the crane documentation for a full reference: http://tsuru-crane.readthedocs.org.

8.4 bs

bs (or big sibling) is a component tsuru component, responsible for reporting information on application containers, this information include application logs, metrics and unit status.

See the bs documentation for a full reference: https://github.com/tsuru/bs#bs.

8.5 tsuru.conf reference

tsuru uses a configuration file in YAML format. This document describes what each option means, and how it should look.

8.5.1 Notation

tsuru uses a colon to represent nesting in YAML. So, whenever this document says something like `key1:key2`, it refers to the value of the `key2` that is nested in the block that is the value of `key1`. For example, `database:url` means:

```
database:
  url: <value>
```

8.5.2 tsuru configuration

This section describes tsuru's core configuration. Other sections will include configuration of optional components, and finally, a full sample file.

HTTP server

tsuru provides a REST API, that supports HTTP and HTTP/TLS (a.k.a. HTTPS). Here are the options that affect how tsuru's API behaves:

listen

`listen` defines in which address tsuru webserver will listen. It has the form <host>:<port>. You may omit the host (example: `:8080`). This setting has no default value.

use-tls

`use-tls` indicates whether tsuru should use TLS or not. This setting is optional, and defaults to "false".

tls:cert-file

`tls:cert-file` is the path to the X.509 certificate file configured to serve the domain. This setting is optional, unless `use-tls` is true.

tls:key-file

`tls:key-file` is the path to private key file configured to serve the domain. This setting is optional, unless `use-tls` is true.

server:read-timeout

`server:read-timeout` is the timeout of reading requests in the server. This is the maximum duration of any request to the tsuru server.

This is useful to avoid leaking connections, in case clients drop the connection before end sending the request. The default value is 0, meaning no timeout.

server:write-timeout

`server:write-timeout` is the timeout of writing responses in the server.

This is useful to avoid leaking connections, in case clients drop the connection before reading the response from tsuru. The default value is 0, meaning no timeout.

disable-index-page

tsuru API serves an index page with some basic instructions on how to use the current target. It's possible to disable this page by setting the `disable-index-page` flag to true. It's also possible to customize which template will be used in the index page, see the next configuration entry for more details.

This setting is optional, and defaults to `false`.

index-page-template

`index-page-template` is the template that will be used for the index page. It must use the Go template syntax, and tsuru will provide the following variables in the context of the template:

- `tsuruTarget`: the target URL of the tsuru API serving the index page
- `userCreate`: a boolean indicating whether user registration is enabled or disabled
- `nativeLogin`: a boolean indicating whether the API is configured to use the native authentication scheme
- `keysEnabled`: a boolean indicating whether the API is configured to manage SSH keys

It will also include a function used for querying configuration values, named `getConfig`. Here is an example of the function usage:

```
<body>
    {{if getConfig "use-tls"}}
    <p>we're safe</p>
    {{else}}
    <p>we're not safe</p>
    {{end}}
</body>
```

This setting is optional. When `index-page-template` is not defined, tsuru will use the default template.

Database access

tsuru uses MongoDB as a database manager to store information like users, machines, containers, etc. You need to describe how tsuru will connect to your database server. Therefore, it's necessary to provide a MongoDB connection string. Database related options are listed below:

database:url

`database:url` is the database connection string. It is a mandatory setting and it has no default value. Examples of strings include basic `127.0.0.1` and more advanced `mongodb://user:password@127.0.0.1:27017/database`. Please refer to MongoDB documentation for more details and examples of connection strings.

database:name

`database:name` is the name of the database that tsuru uses. It is a mandatory setting and has no default value. An example of value is "tsuru".

database:logdb-url

This setting is optional. If `database:logdb-url` is specified, tsuru will use it as the connection string to the MongoDB server responsible for storing application logs. If this value is not set, tsuru will use `database:url` instead.

This setting is useful because tsuru may have to process a very large number of log messages depending on the number of units deployed and applications behavior. Every log message will trigger a insertion in MongoDB and this may negatively impact the database performance. Other measures will be implemented in the future to improve this, but for now, having the ability to use an exclusive database server for logs will help mitigate the negative impact of log writing.

database:logdb-name

This setting is optional. If `database:logdb-name` is specified, tsuru will use it as the database name for storing application logs. If this value is not set, tsuru will use `database:name` instead.

Email configuration

tsuru sends email to users when they request password recovery. In order to send those emails, tsuru needs to be configured with some SMTP settings. Omitting these settings won't break tsuru, but users will not be able to reset their password.

smtp:server

The SMTP server to connect to. It must be in the form <host>:<port>. Example: "smtp.gmail.com:587".

smtp:user

The user to authenticate with the SMTP sever. Currently, tsuru requires authenticated sessions.

smtp:password

The password for authentication within the SMTP server.

Repository configuration

tsuru optionally uses Gandalf to manage git repositories. Gandalf exposes a REST API for repositories management and tsuru needs information about the Gandalf HTTP server endpoint.

repo-manager

`repo-manager` represents the repository manager that tsuru-server should use. For backward compatibility reasons, the default value is "gandalf". Users can disable repository and SSH key management by setting "repo-manager" to "none". For more details, please refer to the repository management page in the documentation.

git:api-server

`git:api-server` is the address of the Gandalf API. It should define the entire address, including protocol and port. Examples of value: `http://localhost:9090` and `https://gandalf.tsuru.io:9595`.

Authentication configuration

tsuru has support for `native` and `oauth` authentication schemes.

The default scheme is `native` and it supports the creation of users in tsuru's internal database. It hashes passwords brcypt. Tokens are generated during authentication and are hashed using SHA512.

The `auth` section also controls whether user registration is on or off. When user registration is off, only admin users are able to create new users.

auth:scheme

The authentication scheme to be used. The default value is `native`, the other supported value is `oauth`.

auth:user-registration

This flag indicates whether user registration is enabled. This setting is optional, and defaults to false.

auth:hash-cost

Required only with `native` chosen as `auth:scheme`.

This number indicates how many CPU time you're willing to give to hashing calculation. It is an absolute number, between 4 and 31, where 4 is faster and less secure, while 31 is very secure and *very* slow.

auth:token-expire-days

Required only with `native` chosen as `auth:scheme`.

Whenever a user logs in, tsuru generates a token for him/her, and the user may store the token. `auth:token-expire-days` setting defines the amount of days that the token will be valid. This setting is optional, and defaults to "7".

auth:max-simultaneous-sessions

tsuru can limit the number of simultaneous sessions per user. This setting is optional, and defaults to "unlimited".

auth:oauth

Every config entry inside `auth:oauth` are used when the `auth:scheme` is set to "oauth". Please check rfc6749 for more details.

auth:oauth:client-id

The client id provided by your OAuth server.

auth:oauth:client-secret

The client secret provided by your OAuth server.

auth:oauth:scope

The scope for your authentication request.

auth:oauth:auth-url

The URL used in the authorization step of the OAuth flow. tsuru CLI will receive this URL and trigger the opening a browser on this URL with the necessary parameters.

During the authorization step, tsuru CLI will start a server locally and set the callback to http://localhost:<port>, if `auth:oauth:callback-port` is set tsuru CLI will use its value as <port>. If `auth:oauth:callback-port` isn't present tsuru CLI will automatically choose an open port.

The callback URL should be registered on your OAuth server.

If the chosen server requires the callback URL to match the same host and port as the registered one you should register "http://localhost:<chosen port>" and set the `auth:oauth:callback-port` accordingly.

If the chosen server is more lenient and allows a different port to be used you should register simply "http://localhost" and leave `auth:oauth:callback-port` empty.

auth:oauth:token-url

The URL used in the exchange token step of the OAuth flow.

auth:oauth:info-url

The URL used to fetch information about the authenticated user. tsuru expects a json response containing a field called `email`.

tsuru will also make call this URL on every request to the API to make sure the token is still valid and hasn't been revoked.

auth:oauth:collection

The database collection used to store valid access tokens. Defaults to "oauth_tokens".

auth:oauth:callback-port

The port used in the callback URL during the authorization step. Check docs for `auth:oauth:auth-url` for more details.

Queue configuration

tsuru uses a work queue for asynchronous tasks.

`queue:*` groups configuration settings for a MongoDB server that will be used as storage for delayed execution of queued jobs.

This queue is used to manage creation and destruction of IaaS machines, but tsuru may start using it in more places in the future.

It's not mandatory to configure the queue, however creating and removing machines using a IaaS provider will not be possible.

queue:mongo-url

Connection url for MongoDB server used to store task information.

queue:mongo-database

Database name used in MongoDB. This value will take precedence over any database name already specified in the connection url.

pubsub

`pubsub` configuration is optional and depends on a redis server instance. It's used only for following application logs (running `tsuru app-log -f`). If this is not configured tsuru will fail when running `tsuru app-log -f`.

Previously the configuration for this redis server was inside `redis-queue:*` keys shown below. Using these keys is deprecated and tsuru will start ignoring them before 1.0 release.

pubsub:redis-host

`pubsub:redis-host` is the host of the Redis server to be used for pub/sub. This settings is optional and defaults to "localhost".

pubsub:redis-port

`pubsub:redis-port` is the port of the Redis server to be used for pub/sub. This settings is optional and defaults to 6379.

pubsub:redis-password

`pubsub:redis-password` is the password of the Redis server to be used for pub/sub. This settings is optional and defaults to "", indicating that the Redis server is not authenticated.

pubsub:redis-db

`pubsub:redis-db` is the database number of the Redis server to be used for pub/sub. This settings is optional and defaults to 3.

pubsub:pool-max-idle-conn

`pubsub:pool-max-idle-conn` is the maximum number of idle connections to redis. Defaults to 20.

pubsub:pool-idle-timeout

`pubsub:pool-idle-timeout` is the number of seconds idle connections will remain in connection pool to redis. Defaults to 300.

pubsub:redis-dial-timeout

`pubsub:redis-dial-timeout` is the number of seconds used as dial timeout. Defaults to 0.1.

pubsub:redis-read-timeout

`pubsub:redis-read-timeout` is the number of seconds used as read timeout. Defaults to 1800 (30 minutes).

pubsub:redis-write-timeout

`pubsub:redis-write-timeout` is the number of seconds used as write timeout. Defaults to 0.5.

redis-queue:host

Deprecated. See `pubsub:redis-host`.

redis-queue:port

Deprecated. See `pubsub:redis-port`.

redis-queue:password

Deprecated. See `pubsub:redis-password`.

redis-queue:db

Deprecated. See `pubsub:redis-db`.

Admin users

tsuru has a very simple way to identify admin users: an admin user is a user that is the member of the admin team, and the admin team is defined in the configuration file, using the `admin-team` setting.

admin-team

`admin-team` is the name of the administration team for the current tsuru installation. All members of the administration team is able to use the `tsuru-admin` command.

Quota management

tsuru can, optionally, manage quotas. Currently, there are two available quotas: apps per user and units per app.

tsuru administrators can control the default quota for new users and new apps in the configuration file, and use `tsuru-admin` command to change quotas for users or apps. Quota management is disabled by default, to enable it, just set the desired quota to a positive integer.

quota:units-per-app

`quota:units-per-app` is the default value for units per-app quota. All new apps will have at most the number of units specified by this setting. This setting is optional, and defaults to "unlimited".

quota:apps-per-user

`quota:apps-per-user` is the default value for apps per-user quota. All new users will have at most the number of apps specified by this setting. This setting is optional, and defaults to "unlimited".

Logging

Tsuru supports three logging flavors, that can be enabled or disabled altogether. The default behavior of tsuru is to send all logs to syslog, but it can also send logs to the standard error stream or a file. It's is possible to use any combination of the three flavors at any time in tsuru configuration (e.g.: write logs both to stderr and syslog, or a file and stderr, or to all of the flavors simultaneously).

There's also the possibility to enable or disable debugging log, via the debug flag.

debug

`false` is the default value, so you won't see any noises on logs, to turn it on set it to true, e.g.: `debug: true`

log:file

Use this to specify a path to a log file. If no file is specified, tsuru-server won't write logs to any file.

log:disable-syslog

`log:disable-syslog` indicates whether tsuru-server should disable the use of syslog. `false` is the default value. If it's `true`, tsuru-server won't send any logs to syslog.

log:syslog-tag

`log:syslog-tag` is the tag that will be attached to every log line. The default value is "tsr".

log:use-stderr

`log:use-stderr` indicates whether tsuru-server should write logs to standard error stream. The default value is `false`.

Routers

As of 0.10.0, all your router configuration should live under entries with the format `routers:<router name>`.

routers:<router name>:type (type: hipache, galeb, vulcand)

Indicates the type of this router configuration. The standard router supported by tsuru is hipache. There is also experimental support for galeb and vulcand).

Depending on the type, there are some specific configuration options available.

routers:<router name>:domain (type: hipache, galeb, vulcand)

The domain of the server running your router. Applications created with tsuru will have a address of `http://<app-name>.<domain>`

routers:<router name>:redis-server (type: hipache)

Redis server used by Hipache router. This same server (or a redis slave of it), must be configured in your hipache.conf file.

routers:<router name>:api-url (type: galeb, vulcand)

The URL for the Galeb or vulcand manager API.

routers:<router name>:username (type: galeb)

Galeb manager username.

routers:<router name>:password (type: galeb)

Galeb manager password.

routers:<router name>:environment (type: galeb)

Galeb manager environment used to create virtual hosts and backend pools.

routers:<router name>:farm-type (type: galeb)

Galeb manager farm type used to create virtual hosts and backend pools.

routers:<router name>:plan (type: galeb)

Galeb manager plan used to create virtual hosts and backend pools.

routers:<router name>:project (type: galeb)

Galeb manager project used to create virtual hosts, backend pools and pools.

routers:<router name>:load-balance-policy (type: galeb)

Galeb manager load balancing policy used to create backend pools.

routers:<router name>:rule-type (type: galeb)

Galeb manager rule type used to create rules.

Hipache

hipache:redis-server

Redis server used by Hipache router. This same server (or a redis slave of it), must be configured in your hipache.conf file.

This setting is deprecated in favor of `routers:<router name>:type = hipache` and `routers:<router name>:redis-server`.

hipache:domain

The domain of the server running your hipache server. Applications created with tsuru will have a address of `http://<app-name>.<hipache:domain>`.

This setting is deprecated in favor of `routers:<router name>:type = hipache` and `routers:<router name>:domain`

Defining the provisioner

tsuru has extensible support for provisioners. A provisioner is a Go type that satisfies the *provision.Provisioner* interface. By default, tsuru will use `DockerProvisioner` (identified by the string "docker"), and now that's the only supported provisioner (Ubuntu Juju was supported in the past but its support has been removed from tsuru).

provisioner

`provisioner` is the string the name of the provisioner that will be used by tsuru. This setting is optional and defaults to "docker".

Docker provisioner configuration

docker:collection

Database collection name used to store containers information.

docker:port-allocator

The choice of port allocator. There are two possible values:

- `docker`: trust Docker to allocate ports. Meaning that whenever a container restarts, the port might change (usually, it changes).
- `tsuru`: leverage port allocation to tsuru, so ports mapped to containers never change.

The default value is "docker".

docker:registry

For tsuru to work with multiple docker nodes, you will need a docker-registry. This should be in the form of `hostname:port`, the scheme cannot be present.

docker:registry-max-try

Number of times tsuru will try to send a image to registry.

docker:registry-auth:username

The username used for registry authentication. This setting is optional, for registries with authentication disabled, it can be omitted.

docker:registry-auth:password

The password used for registry authentication. This setting is optional, for registries with authentication disabled, it can be omitted.

docker:registry-auth:email

The email used for registry authentication. This setting is optional, for registries with authentication disabled, it can be omitted.

docker:repository-namespace

Docker repository namespace to be used for application and platform images. Images will be tagged in docker as <docker:repository-namespace>/<platform-name> and <docker:repository-namespace>/<app-name>

docker:bs:image

`docker:bs:image` is the name of the Docker image to be used to create big sibling containers. The default value is "tsuru/bs", which represents the official image hosted at Docker Hub, maintained by the tsuru team.

docker:bs:socket

`docker:bs:socket` is the path to the Unix socket in the Docker host. This should be configured so bs can connect to Docker via socket instead of TCP. This is an optional setting, when omitted, bs will talk to the Docker API using the TCP endpoint.

docker:bs:syslog-port

`docker:bs:syslog-port` is the port in the Docker node that will be used by the bs container for collecting logs. The default value is 1514.

docker:max-workers

Maximum amount of threads to be created when starting new containers, so tsuru doesn't start too much threads in the process of starting 1000 units, for instance. Defaults to 0 which means unlimited.

docker:router

Default router to be used to distribute requests to units. This should be the name of a router configured under the `routers:<name>` key, see *routers*.

For backward compatibility reasons, the value `hipache` is also supported, and it will use either configuration available under `router:hipache:*` or `hipache:*`, in this order.

Note that as of 0.10.0, routers may be associated to plans, if when creating an application the chosen plan has a router value it will be used instead of the value set in `docker:router`.

The router defined in `docker:router` will only be used if the chosen plan doesn't specify one.

docker:deploy-cmd

The command that will be called in your platform when a new deploy happens. The default value for platforms supported in tsuru's basebuilder repository is `/var/lib/tsuru/deploy`.

docker:security-opts

This setting describes a list of security options that will be passed to containers. This setting must be a list, and has no default value. If one wants to specify just one value, it's still needed to use the list notation:

```
docker:
  ...
  security-opts:
    - apparmor:PROFILE
```

For more details on the available options, please refer to the Docker documentation: <https://docs.docker.com/reference/run/#security-configuration>.

docker:segregate

Deprecated. As of tsuru 0.11.1, using segregate scheduler is the default setting. See Segregate Scheduler for details.

docker:scheduler:total-memory-metadata

This value describes which metadata key will describe the total amount of memory, in bytes, available to a docker node.

docker:scheduler:max-used-memory

This should be a value between 0.0 and 1.0 which describes which fraction of the total amount of memory available to a server should be reserved for app units.

The amount of memory available is found based on the node metadata described by `docker:scheduler:total-memory-metadata` config setting.

If this value is set, tsuru will try to find a node with enough unreserved memory to fit the creation of new units, based on how much memory is required by the plan used to create the application. If no node with enough unreserved memory is found, tsuru will ignore memory restrictions and let the scheduler choose any node.

This setting, along with `docker:scheduler:total-memory-metadata`, are also used by node auto scaling. See node auto scaling for more details.

docker:cluster:storage

This setting has been removed. You shouldn't define it anymore, the only storage available for the docker cluster is now `mongodb`.

docker:cluster:mongo-url

Connection URL to the mongodb server used to store information about the docker cluster.

docker:cluster:mongo-database

Database name to be used to store information about the docker cluster.

docker:run-cmd:bin

The command that will be called on the application image to start the application. The default value for platforms supported in tsuru's basebuilder repository is `/var/lib/tsuru/start`.

docker:run-cmd:port

The tcp port that will be exported by the container to the node network. The default value expected by platforms defined in tsuru's basebuilder repository is `8888`.

docker:user

The user tsuru will use to start the container. The value expected for basebuilder platforms is `ubuntu`.

docker:healing:heal-nodes

Boolean value that indicates whether tsuru should try to heal nodes that have failed a specified number of times. Healing nodes is only available if the node was created by tsuru itself using the IaaS configuration. Defaults to `false`.

docker:healing:active-monitoring-interval

Number of seconds between calls to <server>/_ping in each one of the docker nodes. If this value is 0 or unset tsuru will never call the ping URL. Defaults to 0.

docker:healing:disabled-time

Number of seconds tsuru disables a node after a failure. This setting is only valid if `heal-nodes` is set to `true`. Defaults to 30 seconds.

docker:healing:max-failures

Number of consecutive failures a node should have before triggering a healing operation. Only valid if `heal-nodes` is set to `true`. Defaults to 5.

docker:healing:wait-new-time

Number of seconds tsuru should wait for the creation of a new node during the healing process. Only valid if `heal-nodes` is set to `true`. Defaults to 300 seconds (5 minutes).

docker:healing:heal-containers-timeout

Number of seconds a container should be unresponsive before triggering the recreation of the container. A container is deemed unresponsive if it doesn't call the set unit status URL (/apps/{app}/units/{unit}) with a `started` status. If this value is 0 or unset tsuru will never try to heal unresponsive containers. Defaults to 0.

docker:healing:events_collection

Collection name in mongodb used to store information about triggered healing events. Defaults to `healing_events`.

docker:healthcheck:max-time

Maximum time in seconds to wait for deployment time health check to be successful. Defaults to 120 seconds.

docker:image-history-size

Number of images available for rollback using `tsuru app-deploy-rollback`. tsuru will try to delete older images, but it may not be able to due to it being used as a layer to a newer image. tsuru will keep trying to remove these old images until they are not used as layers anymore. Defaults to 10 images.

docker:auto-scale:enabled

Enable node auto scaling. See node auto scaling for more details. Defaults to false.

docker:auto-scale:wait-new-time

Number of seconds tsuru should wait for the creation of a new node during the scaling up process. Defaults to 300 seconds (5 minutes).

docker:auto-scale:group-by-metadata

Name of the metadata present in nodes that will be used for grouping nodes into clusters. See node auto scaling for more details. Defaults to empty (all nodes belong the the same cluster).

docker:auto-scale:metadata-filter

Value of the metadata specified by *docker:auto-scale:group-by-metadata*. If this is set, tsuru will only run auto scale algorithms for nodes in the cluster defined by this value.

docker:auto-scale:max-container-count

Maximum number of containers per node, for count based scaling. See node auto scaling for more details.

docker:auto-scale:prevent-rebalance

Prevent rebalancing from happening when adding new nodes, or if a rebalance is needed. See node auto scaling for more details.

docker:auto-scale:run-interval

Number of seconds between two periodic runs of the auto scaling algorithm. Defaults to 3600 seconds (1 hour).

docker:auto-scale:scale-down-ratio

Ratio used when scaling down. Must be greater than 1.0. See node auto scaling for more details. Defaults to 1.33.

8.5.3 IaaS configuration

tsuru uses IaaS configuration to automatically create new docker nodes and adding them to your cluster when using `docker-node-add` command. See adding nodes for more details about how to use this command.

Attention: You should configure *queue* to be able to use IaaS.

General settings

iaas:default

The default IaaS tsuru will use when calling `docker-node-add` without specifying `iaas=<iaas_name>` as a metadata. Defaults to `ec2`.

iaas:node-protocol

Which protocol to use when accessing the docker api in the created node. Defaults to `http`.

iaas:node-port

In which port the docker API will be accessible in the created node. Defaults to `2375`.

iaas:collection

Collection name on database containing information about created machines. Defaults to `iaas_machines`.

EC2 IaaS

iaas:ec2:key-id

Your AWS key id.

iaas:ec2:secret-key

Your AWS secret key.

iaas:ec2:user-data

A url for which the response body will be sent to ec2 as user-data. Defaults to a script which will run tsuru now installation.

iaas:ec2:wait-timeout

Number of seconds to wait for the machine to be created. Defaults to 300 (5 minutes).

CloudStack IaaS

iaas:cloudstack:api-key

Your api key.

iaas:cloudstack:secret-key

Your secret key.

iaas:cloudstack:url

The url for the cloudstack api.

iaas:cloudstack:user-data

A url for which the response body will be sent to cloudstack as user-data. Defaults to a script which will run tsuru now installation.

iaas:cloudstack:wait-timeout

Number of seconds to wait for the machine to be created. Defaults to 300 (5 minutes).

Custom IaaS

You can define a custom IaaS based on an existing provider. Any configuration keys with the format `iaas:custom:<name>` will create a new IaaS with `name`.

iaas:custom:<name>:provider

The base provider name, it can be one of the supported providers: `cloudstack` or `ec2`.

iaas:custom:<name>:<any_other_option>

This will overwrite the value of `iaas:<provider>:<any_other_option>` for this IaaS. As an example, having the configuration below would allow you to call `tsuru-admin docker-node-add iaas=region1_cloudstack ...`:

```
iaas:
    custom:
        region1_cloudstack:
            provider: cloudstack
            url: http://region1.url/
            secret-key: mysecretkey
    cloudstack:
        api-key: myapikey
```

8.5.4 Sample file

Here is a complete example:

```
listen: "0.0.0.0:8080"
debug: true
host: http://<machine-public-addr>:8080 # This port must be the same as in the "listen" conf
admin-team: admin
auth:
    user-registration: true
    scheme: native
database:
    url: <your-mongodb-server>:27017
    name: tsurudb
pubsub:
    redis-host: <your-redis-server>
    redis-port: 6379
queue:
    mongo-url: <your-mongodb-server>:27017
    mongo-database: queuedb
git:
    api-server: http://<your-gandalf-server>:8000
provisioner: docker
docker:
    router: hipache
    collection: docker_containers
    repository-namespace: tsuru
    deploy-cmd: /var/lib/tsuru/deploy
    cluster:
        storage: mongodb
        mongo-url: <your-mongodb-server>:27017
        mongo-database: cluster
    run-cmd:
        bin: /var/lib/tsuru/start
        port: "8888"
routers:
    hipache:
        type: hipache
        domain: <your-hipache-server-ip>.xip.io
        redis-server: <your-redis-server-with-port>
```

8.6 API reference

8.6.1 1. Endpoints

1.1 Apps

List apps

- Method: GET
- Endpoint: /apps
- Format: JSON

Returns 200 in case of success, and JSON in the body of the response containing the app list.

Example:

```
GET /apps HTTP/1.1
Content-Length: 82
[{"Ip":"10.10.10.10","Name":"app1","Units":[{"Name":"app1/0","State":"started"}]}]
```

Info about an app

- Method: GET
- Endpoint: /apps/<appname>
- Format: JSON

Returns 200 in case of success, and a JSON in the body of the response containing the app content.

Example:

```
GET /apps/myapp HTTP/1.1
Content-Length: 284
{"Name":"app1","Framework":"php","Repository":"git@git.com:php.git","State":"dead","Units":[{"Ip":"1(
```

Remove an app

- Method: DELETE
- Endpoint: /apps/<appname>

Returns 200 in case of success.

Example:

```
DELETE /apps/myapp HTTP/1.1
```

Create an app

- Method: POST
- Endpoint: /apps
- Format: JSON

Returns 200 in case of success, and JSON in the body of the response containing the status and the URL for Git repository.

Example:

```
POST /apps HTTP/1.1
{"status":"success", "repository_url":"git@tsuru.mycompany.com:ble.git"}
```

Restart an app

- Method: GET
- Endpoint: /apps/<appname>/restart

Returns 200 in case of success.

Example:

```
GET /apps/myapp/restart HTTP/1.1
```

Get app environment variables

- Method: GET
- Endpoint: /apps/<appname>/env

Returns 200 in case of success, and JSON in the body returning a dictionary with environment names and values.

Example:

```
GET /apps/myapp/env HTTP/1.1
[{"name": "DATABASE_HOST", "value": "localhost", "public": true}]
```

Set an app environment

- Method: POST
- Endpoint: /apps/<appname>/env

Returns 200 in case of success.

Example:

```
POST /apps/myapp/env HTTP/1.1
```

Execute a command

- Method: POST
- Endpoint: /apps/<appname>/run?once=true

Returns 200 in case of success.

Where:

- *once* is a boolean and indicates if the command will run just in an unit(once=true) or all of them(once=false). This parameter is not required, and the default is false.

Example:

```
POST /apps/myapp/run HTTP/1.1
ls -la
```

Remove one or more environment variables from an app

- Method: DELETE
- Endpoint: /apps/<appname>/env

Returns 200 in case of success.

Example:

```
DELETE /apps/myapp/env HTTP/1.1
```

Swap the address of two apps

- Method: PUT
- Endpoint: /swap?app1=appname&app2=anotherapp

Returns 200 in case of success.

Example:

```
PUT /swap?app1=myapp&app2=anotherapp
```

Get the logs of an app

- Method: GET
- Endpoint: /apps/appname/log?lines=10&source=web&unit=abc123

Returns 200 in case of success. Returns 404 if app is not found.

Where:

- *lines* is the number of the log lines. This parameter is required.
- *source* is the source of the log, like *tsuru* (tsuru API) or a process.
- *unit* is the *id* of an unit.

Example:

```
GET /apps/myapp/log?lines=20&source=web&unit=83535b503c96
Content-Length: 142
[{"Date":"2014-09-26T00:26:30.036Z","Message":"Booting worker with pid: 53","Source":"web","AppName"
```

List available pools

- Method: GET
- Endpoint: /pools

Returns 200 in case of success.

Example:

```
GET /pools
[{"Team":"team1","Pools":["pool1","pool2"]},{"Team":"team2","Pools":["pool3"]}]
```

Change the pool of an app

- Method: POST
- Endpoint: /apps/<appname>/pool

Returns 200 in case of success. Returns 404 if app is not found.

Example:

```
POST /apps/myapp/pool
```

1.2 Services

List services

- Method: GET
- Endpoint: /services
- Format: JSON

Returns 200 in case of success.

Example:

```
GET /services HTTP/1.1
Content-Length: 67
{"service": "mongodb", "instances": ["my_nosql", "other-instance"]}
```

Create a new service

- Method: POST
- Endpoint: /services
- Format: yaml
- Body: a yaml with the service metadata.

Returns 200 in case of success. Returns 403 if the user is not a member of a team. Returns 500 if the yaml is invalid. Returns 500 if the service name already exists.

Example:

```
POST /services HTTP/1.1
id: some_service
endpoint:
  production: someservice.com
```

Remove a service

- Method: DELETE
- Endpoint: /services/<servicename>

Returns 204 in case of success. Returns 403 if user has not access to the server. Returns 403 if service has instances. Returns 404 if service is not found.

Example:

```
DELETE /services/mongodb HTTP/1.1
```

Update a service

- Method: PUT
- Endpoint: /services
- Format: yaml
- Body: a yaml with the service metadata.

Returns 200 in case of success. Returns 403 if the user is not a member of a team. Returns 500 if the yaml is invalid. Returns 500 if the service name already exists.

Example:

```
PUT /services HTTP/1.1
id: some_service
endpoint:
  production: someservice.com
```

Get info about a service

- Method: GET
- Endpoint: /services/<servicename>
- Format: JSON

Returns 200 in case of success. Returns 404 if the service does not exists.

Example:

```
GET /services/mongodb HTTP/1.1
[{"Name": "my-mongo", "Teams": ["myteam"], "Apps": ["myapp"], "ServiceName": "mongodb"}]
```

Get service documentation

- Method: GET
- Endpoint: /services/<servicename>/doc
- Format: text

Returns 200 in case of success. Returns 404 if the service does not exists.

Example:

```
GET /services/mongodb/doc HTTP/1.1
Mongodb exports the ...
```

Update service documentation

- Method: PUT
- Endpoint: /services/<servicename>/doc
- Format: text
- Body: text with the documentation

Returns 200 in case of success. Returns 404 if the service does not exists.

Example:

```
PUT /services/mongodb/doc HTTP/1.1
Body: Mongodb exports the ...
```

Grant access to a service

- Method: PUT
- Endpoint: /services/<servicename>/<teamname>

Returns 200 in case of success. Returns 404 if the service does not exists.

Example:

```
PUT /services/mongodb/cobrateam HTTP/1.1
```

Revoke access from a service

- Method: DELETE
- Endpoint: /services/<servicename>/<teamname>

Returns 200 in case of success. Returns 404 if the service does not exists.

Example:

```
DELETE /services/mongodb/cobrateam HTTP/1.1
```

1.3 Service instances

Add a new service instance

- Method: POST
- Endpoint: /services/instances
- Body: *{"name": "mymysql", "service_name": "mysql"}*

Returns 200 in case of success. Returns 404 if the service does not exists.

Example:

```
POST /services/instances HTTP/1.1
{"name": "mymysql", "service_name": "mysql"}
```

Remove a service instance

- Method: DELETE
- Endpoint: /services/instances/<serviceinstancename>

Returns 200 in case of success. Returns 404 if the service does not exists.

Example:

```
DELETE /services/instances/mymysql HTTP/1.1
```

Bind a service instance to an app

- Method: PUT
- Endpoint: /services/instances/<serviceinstancename>/<appname>
- Format: JSON

Returns 200 in case of success, and JSON with the environment variables to be exported in the app environ. Returns 403 if the user has not access to the app. Returns 404 if the application does not exists. Returns 404 if the service instance does not exists.

Example:

```
PUT /services/instances/mymysql/myapp HTTP/1.1
Content-Length: 29
{"DATABASE_HOST":"localhost"}
```

Unbind a service instance from an app

- Method: DELETE
- Endpoint: /services/instances/<serviceinstancename>/<appname>

Returns 200 in case of success. Returns 403 if the user has not access to the app. Returns 404 if the application does not exists. Returns 404 if the service instance does not exists.

Example:

```
DELETE /services/instances/mymysql/myapp HTTP/1.1
```

List all services and your instances

- Method: GET
- Endpoint: /services/instances?app=appname
- Format: JSON

Returns 200 in case of success and a JSON with the service list.

Where:

- *app* is the name an app you want to use as filter. If defined only instances bound to this app will be returned. This parameter is optional.

Example:

```
GET /services/instances HTTP/1.1
Content-Length: 52
[{"service": "redis", "instances": ["redis-globo"]}]
```

Get an info about a service instance

- Method: GET
- Endpoint: /services/instances/<serviceinstancename>
- Format: JSON

Returns 200 in case of success and a JSON with the service instance data. Returns 404 if the service instance does not exists.

Example:

```
GET /services/instances/mymysql HTTP/1.1
Content-Length: 71
{"name": "mongo-1", "servicename": "mongodb", "teams": [], "apps": []}
```

service instance status

- Method: GET
- Endpoint: /services/instances/<serviceinstancename>/status

Returns 200 in case of success.

Example:

```
GET /services/instances/mymysql/status HTTP/1.1
```

Grant access to a service instance

- Method: PUT
- Endpoint: /services/instances/permission/<servicename>/<teamname>

Returns 200 in case of success. Returns 404 if the service does not exists.

Example:

```
PUT /services/instances/permission/mongodb-instance/cobrateam HTTP/1.1
```

Revoke access from a service instance

- Method: DELETE
- Endpoint: /services/instances/permission/<servicename>/<teamname>

Returns 200 in case of success. Returns 404 if the service does not exists.

Example:

```
DELETE /services/instances/permission/mongodb-instance/cobrateam HTTP/1.1
```

1.4 Quotas

Get quota info of a user

- Method: GET
- Endpoint: /quota/<user>
- Format: JSON

Returns 200 in case of success, and JSON with the quota info.

Example:

```
GET /quota/wolverine HTTP/1.1
Content-Length: 29
{"items": 10, "available": 2}
```

1.5 Healers

List healers

- Method: GET
- Endpoint: /healers
- Format: JSON

Returns 200 in case of success, and JSON in the body with a list of healers.

Example:

```
GET /healers HTTP/1.1
Content-Length: 35
[{"app-heal": "http://healer.com"}]
```

Execute healer

- Method: GET
- Endpoint: /healers/<healer>

Returns 200 in case of success.

Example:

```
GET /healers/app-heal HTTP/1.1
```

1.6 Platforms

List platforms

- Method: GET
- Endpoint: /platforms
- Format: JSON

Returns 200 in case of success, and JSON in the body with a list of platforms.

Example:

```
GET /platforms HTTP/1.1
Content-Length: 67
[{Name: "python"},{Name: "java"},{Name: "ruby20"},{Name: "static"}]
```

1.7 Users

Create a user

- Method: POST
- Endpoint: /users
- Body: *{"email":"nobody@globo.com","password":"123456"}*

Returns 200 in case of success. Returns 400 if the JSON is invalid. Returns 400 if the email is invalid. Returns 400 if the password characters length is less than 6 and greater than 50. Returns 409 if the email already exists.

Example:

```
POST /users HTTP/1.1
Body: `{"email":"nobody@globo.com","password":"123456"}`
```

Reset password

- Method: POST
- Endpoint: /users/<email>/password?token=token

Returns 200 in case of success. Returns 404 if the user is not found.

The token parameter is optional.

Example:

```
POST /users/user@email.com/password?token=1234 HTTP/1.1
```

Login

- Method: POST
- Endpoint: /users/<email>/tokens
- Body: *{"password":"123456"}*

Returns 200 in case of success. Returns 400 if the JSON is invalid. Returns 400 if the password is empty or nil. Returns 404 if the user is not found.

Example:

```
POST /users/user@email.com/tokens HTTP/1.1
{"token":"e275317394fb099f62b3993fd09e5f23b258d55f"}
```

Logout

- Method: DELETE
- Endpoint: /users/tokens

Returns 200 in case of success.

Example:

```
DELETE /users/tokens HTTP/1.1
```

Info about the current user

- Method: GET
- Endpoint: /users/info

Returns 200 in case of success, and a JSON with information about the current user.

Example:

```
GET /users/info HTTP/1.1
{"Email":"myuser@company.com","Teams":["frontend","backend","sysadmin","full stack"]}
```

Change password

- Method: PUT
- Endpoint: /users/password
- Body: *{"old":"123456","new":"654321"}*

Returns 200 in case of success. Returns 400 if the JSON is invalid. Returns 400 if the old or new password is empty or nil. Returns 400 if the new password characters length is less than 6 and greater than 50. Returns 403 if the old password does not match with the current password.

Example:

```
PUT /users/password HTTP/1.1
Body: `{"old":"123456","new":"654321"}`
```

Remove a user

- Method: DELETE
- Endpoint: /users

Returns 200 in case of success.

Example:

```
DELETE /users HTTP/1.1
```

Add public key to user

- Method: POST
- Endpoint: /users/keys
- Body: *{"key":"my-key"}*

Returns 200 in case of success.

Example:

```
POST /users/keys HTTP/1.1
Body: `{"key":"my-key"}`
```

Remove public key from user

- Method: DELETE
- Endpoint: /users/keys
- Body: *{"key":"my-key"}*

Returns 200 in case of success.

Example:

```
DELETE /users/keys HTTP/1.1
Body: `{"key":"my-key"}`
```

Show API key

- Method: GET
- Endpoint: /users/api-key
- Format: JSON

Returns 200 in case of success, and JSON in the body with the API key.

Example:

```
GET /users/api-key HTTP/1.1
Body: `{"token": "e275317394fb099f62b3993fd09e5f23b258d55f", "users": "user@email.com"}`
```

Regenerate API key

- Method: POST
- Endpoint: /users/api-key

Returns 200 in case of success.

Example:

```
POST /users/api-key HTTP/1.1
```

1.8 Teams

List teams

- Method: GET
- Endpoint: /teams
- Format: JSON

Returns 200 in case of success, and JSON in the body with a list of teams.

Example:

```
GET /teams HTTP/1.1
Content-Length: 22
[{"name": "teamname"}]
```

Info about a team

- Method: GET
- Endpoint: /teams/<teamname>
- Format: JSON

Returns 200 in case of success, and JSON in the body with the info about a team.

Example:

```
GET /teams/teamname HTTP/1.1
{"name": "teamname", "users": ["user@email.com"]}
```

Add a team

- Method: POST
- Endpoint: /teams

Returns 200 in case of success.

Example:

```
POST /teams HTTP/1.1
{"name": "teamname"}
```

Remove a team

- Method: DELETE
- Endpoint: /teams/<teamname>

Returns 200 in case of success.

Example:

```
DELELE /teams/myteam HTTP/1.1
```

Add user to team

- Method: PUT
- Endpoint: /teams/<teanmaname>/<username>

Returns 200 in case of success.

Example:

```
PUT /teams/myteam/myuser HTTP/1.1
```

Remove user from team

- Method: DELETE
- Endpoint: /teams/<teanmaname>/<username>

Returns 200 in case of success.

Example:

```
DELETE /teams/myteam/myuser HTTP/1.1
```

1.9 Deploy

Deploy list

- Method: GET
- Endpoint: /deploys?app=appname&service=servicename
- Format: JSON

Returns 200 in case of success, and JSON in the body of the response containing the deploy list.

Where:

- *app* is a *app* name.
- *service* is a *service* name.

Example:

```
GET /deploys HTTP/1.1
[{"Ip":"10.10.10.10","Name":"app1","Units":[{"Name":"app1/0","State":"started"}]}]
[{"ID":"543c20a09e7aea60156191c0","App":"myapp","Timestamp":"2013-11-01T00:01:00-02:00","Duration":2
```

Get info about a deploy

- Method: GET
- Format: JSON
- Endpoint: /deploys/:deployid

Returns 200 in case of success. Returns 404 if deploy is not found.

Example:

```
GET /deploys/12345
{"ID":"54ff355c283dbed9868f01fb","App":"tsuru-dashboard","Timestamp":"2015-03-10T15:18:04.301-03:00",
```

1.10 Metadata

Info about Tsuru API

- Method: GET
- Endpoint: /info
- Format: JSON

Returns 200 in case of success, and JSON in the body of the response containing the metadata.

Example:

```
GET /info HTTP/1.1
{"version": "1.0"}
```

Basic healthcheck of Tsuru API server

- Method: GET
- Endpoint: /healthcheck/
- Format: text

Always returns 200 and text body of `WORKING`.

Example:

```
GET /healthcheck/ HTTP/1.1
WORKING
```

Full healthcheck of all Tsuru components

- Method: GET
- Endpoint: /healthcheck/?check=all
- Format: text

Returns 200 when all components have a status of `WORKING`. Returns 500 if any component does not have a status of `WORKING`. Body always contains text with status and time to complete check for each component.

Example:

```
GET /healthcheck/?check=all HTTP/1.1
MongoDB: WORKING (643.81µs)
Router Hipache: WORKING (845.457µs)
docker-registry: WORKING (1.954069ms)
Gandalf: WORKING (1.787768ms)
```

CHAPTER 9

Frequently Asked Questions

This document is an attempt to explain concepts you'll face when deploying and managing applications using tsuru. To request additional explanations you can open an issue on our issue tracker, talk to us at #tsuru @ freenode.net or open a thread on our mailing list.

9.1 How do environment variables work?

All configurations in tsuru are handled by the use of environment variables. If you need to connect with a third party service, e.g. twitter's API, you are probably going to need some extra configurations, like client_id. In tsuru, you can export those as environment variables, visible only by your application's processes.

When you bind your application into a service, most likely you'll need to communicate with that service in some way. Services can export environment variables by telling tsuru what they need, so whenever you bind your application with a service, its API can return environment variables for tsuru to export on your application's units.

9.2 How does the quota system work?

Quotas are handled per application and user. Every user has a quota number for applications. For example, users may have a default quota of 2 applications, so whenever a user tries to create more than two applications, he/she will receive a quota exceeded error. There are also per applications quota. This one limits the maximum number of units that an application may have.

9.3 How does routing work?

tsuru has a router interface, which makes it extremely easy to change the way routing works with any provisioner. There are two ready-to-go routers: one using hipache and another with galeb.

Note: as of 0.10.0 version **tsuru** supports more than one router. You can have a default router, configured by "docker:router" and you can define a custom router by plan

9.4 How are Git repositories managed?

tsuru uses Gandalf to manage git repositories. Every time you create an application, tsuru will ask Gandalf to create a related git bare repository for you to push in.

This is the remote tsuru gives you when you create a new app. Everytime you perform a git push, Gandalf intercepts it, check if you have the required authorization to write into the application's repository, and then lets the push proceeds or returns an error message.

Note: For *tsuru-admin*, *tsuru* and *crane* release notes, check GitHub release history:

- crane: https://github.com/tsuru/crane/releases
- tsuru: https://github.com/tsuru/tsuru-client/releases
- tsuru-admin: https://github.com/tsuru/tsuru-admin/releases

CHAPTER 10

Release notes

Release notes for the official tsuru releases. Each release note will tell you what's new in each version.

10.1 tsurud (tsuru server daemon)

Warning: tsurud used to be called tsr, the name changed in the 0.12.0 release.

10.1.1 tsurud 0.12.3 release notes

Welcome to tsurud 0.12.3!

tsurud 0.12.3 includes *bug fixes* and some *improvements* on unstable network environments.

Improvements

- On some unstable network environments it was possible for a deploy to remain frozen while running Attach and Wait operations on the docker node. This can happen after a network partition where the connection was severed without FIN or RST being sent from on end to the other.

  This problem was solved in two different ways. First TCP keepalive was enabled for all connections with the Docker API. This way if there are any problems severing the connection, the keepalive probe will hopefully receive RST as an answer when the connectivity with the remote server is re-established, closing the connection on our end.

  As a failsafe, while tsuru is blocked on Attach and Wait requests it will also keep polling Docker for the current container state. If the container is stopped it means that the Attach and Wait operations should have ended. At this moment tsuru will resume the deploy process and ignore the output from Attach and Wait.
- Use the KeepAliveWriter across all streaming handlers in the API, so the API is able to cope with small timeouts in the network.
- Add a service level proxy so service APIs can have management plugins. This proxy endpoint checks the permission of the user as an admin of the service. The other proxy endpoint checks the user permission in the service instance.

Bug fixes

- Fix bug in `/units/status` route that is called by bs containers. The bug caused this route to return a 500 error if the request included containers with the status `building` in tsuru's database.
- Fix error message in the docker-node-update handler when it's called with an invalid name (issue #1207).
- Fix bug in Procfile parsing in the API. We used to parse it as YAML, but a Procfile is not really an YAML.
- Properly manage repository permissions in Gandalf after running app-set-team-owner (issue #1270).
- Fix quota management for units in applications (issue #1279).

10.1.2 tsurud 0.12.2 release notes

Welcome to tsurud 0.12.2!

tsurud 0.12.2 includes *bug fixes* related to application environment variables.

Bug fixes

Two different bugs prevented commands setting and unsetting environment variables for an application from working correctly. This release also depends on updating platforms to use tsuru-unit-agent version 0.4.5.

- The first bug prevented `env-unset` from working because environment variables were being committed in the application image during the deploy. This way, it wasn't possible to unset a variable because even if they were not used when starting a new container the image would include them.
- The second bug prevented `env-set` from overriding the value of a previously set environment variable after at least one deploy happened with the first value set.

  This bug happened because during deploy tsuru would write a file called `apprc` including all environment variables available during the deploy and this file would then be loaded in the application environment, overriding environment variables used to start the container.

  This file was only needed by tsuru versions before 0.12.0 and the solution was simply not to add application environment variables to this file anymore if tsuru server is greater than or equal to 0.12.0.

10.1.3 tsurud 0.12.1 release notes

Welcome to tsurud 0.12.1!

tsurud 0.12.1 includes *bug fixes* and *improvements* in the management of the tsuru host agent (bs).

General improvements

- Improve node registering process: now, when the creation of the bs container fails, we do not destroy managed hosts, but rather mark them as "waiting". tsuru already ensures that bs is running in the node before executing other operations.
- Use "ready" as the status of nodes running bs. In case everything goes fine during node creation/registration, tsuru will now mark the node as "ready" instead of "waiting".
- Use "tsuru/bs:v1" as the default bs image. It's possible to use "tsuru/bs" to get the old behavior back, or even "tsuru/bs:latest" to seat on the bleeding edge of bs.

Bug fixes

- Fix race condition between bs status reporting and the deployment process, preventing bs from destroying containers that are still being deployed.
- Fix application token leaking in the OAuth authentication scheme.
- Prevent the removal of swapped applications to avoid router inconsistencies.
- Fix inconsistency in the Galeb router: it didn't handle removal properly, leading to inconsistencies in the router after running `tsuru app-plan-change`.
- Fix swapping applications using hipache router. There was a bug that allowed only the first swap and wouldn't allow swapping back.

10.1.4 tsurud 0.12.0 release notes

Welcome to tsurud 0.12.0!

These release notes cover the *new features*, *bug fixes*, *general improvements* and *backward incompatible changes* you'll want to be aware of when upgrading from tsr 0.11.2 or older versions.

Main new features

- Lean containers: this is definitely the big feature of this release. With lean containers, we've dropped Circus, making application images smaller, and containers faster. Improving resource usage.

  Application containers won't run tsuru-unit-agent anymore either. It's still used during the deployment process, but it's not competing with the application process anymore.

  Instead of having one agent inside each unit, Docker nodes will now have one agent collecting information about containers running in the node. This agent is named bs. The default behavior of tsuru is to create the bs container before running operation in the node. It should work out-of-the-box after the update, but you can tune *bs configuration*, customizing the Docker image for running it or configuring it to use Unix socket instead of TCP for Docker API communication (which is safer).

  tsuru will create and manage at least one container per Procfile entry. Users are now able to manage the amount of units for each process.

  Latest tsuru-admin release includes commands for managing bs configuration.

  See issues #647 and #1136 for more details.

- There are now three kinds of pools: by team, public and default. Team's pool are segregated by teams, and cloud administrator should set teams in this pool manually. This pool are just accessible by team's members.

  Public pools are accessible by any user. It can be used to segregate machines that have specific hardware.

  Default pool are for experimentation and low profile apps, like service dashboard and "in development" apps. This is the old fallback pool, but with an explicit flag.

- New router available: vulcand (thanks Dan Carley). Vulcand is a powerful reverse proxy, with SNI based TLS support. This is the first step on being able to configure TLS on applications (see issue #1206).

  It's now possible to choose between Hipache, Galeb (which is still partially open source) and Vulcand.

- Users are now able to change the plan of an application. tsuru will handle changes in the router and in other plan-defined application resources (i.e. memory, swap and CPU shares) #1181

- Introduce a custom port allocator on tsuru. This allocator replaces the default port allocation provided by Docker, offering a way of persisting the port of a container after restarts.

  The motivation behind this feature is making sure the host port mapped to one container never changes, even after restarting docker daemon or rebooting the host. This way, we can always be sure that routers are pointing to a valid address.

  The default behavior is to stick to the Docker allocator, please refer to the *port-allocator configuration documentation* for instructions on how to choose the tsuru allocator.

  This is related to issue #1072.

Bug fixes

- Properly handle suffixes when adding a CNAME to an application (thanks Leandro Souza). #1215
- Improve safety in app-restart and other containers related operations. #1188
- Admin users can now delete any teams. #1232
- Prevent service instances orphaning by not allowing a team that is the owner of a service instance to be removed. #1236
- Properly handle key overriding on key management functions. Previously, when a user added a new key reusing a name, tsuru created the new key with the given name and body, letting the old body as an orphan key, making it impossible to remove the old key or associate it to another user. #1249
- Unbind is now atomic, meaning that it's safer to service administrators to trust on tsuru service operations being all-or-nothing. #1253
- Fix error message on app-create when pool doesn't exist. #1257

Other improvements

- Now tsuru doesn't try to start stopped/errored containers when containers move. #1186
- app-shell now uses WebSocket for communication between the tsuru client and the API. This allows app-shell to be used behind proxies that support WebSocket (e.g. nginx). For more details, see #1162.
- tsuru will always use the segregate scheduler, the round robin scheduler has been disabled. In order to get a similar behavior, cloud admins can create a single pool and set it as the default pool, so users don't need to choose the pool on `app-create`.
- tsuru is now compatible with Docker 1.8.x. There was a small change in the Docker API, changing the way of handling mount points, which affected shared file systems.
- Node auto-scaling now support multi-step scaling, meaning that when scaling up or down, it might add or remove multiple nodes at once. This reduces lock content on applications and the amount of containers rebalance runnings.
- Support for Docker Registry API v2 (also known as Docker Distribution).
- Application logs are now collected via WebSocket as well. Each Docker node connects to the tsuru API once, and then streams logs from all containers in the node.
- Change application tokens so they never expire.
- The EC2 IaaS now supports tagging. #1094
- Add configuration options for timeouts in the Redis pubsub connection (use for real time logging, a.k.a. `tsuru app-log -f`).

- Add a heartbeat for keeping connections open during platform-add and platform-update (thanks Richard Knop).
- Improve error reporting in the user API (thanks Dan Hilton).
- Change the behavior of unit-remove and app-remove handlers so they don't run in background.
- Enforce memory limits on Docker nodes when auto-scale is disabled. Now, whenever node auto-scaling is disabled, tsuru will enforce the max memory policy because this will trigger an error and someone will have to manually add a new node to allow new units to be created. #1251
- `docker-node-remove` command now rebalance all containers in removed host. You also have a flag, `--no-rebalance`, to not rebalance thes containers. #1246
- Add `--disable` flag in `docker-node-update` command. This flag tag your node as disabled in cluster. #1246
- General improvements in the documentation:
  - add documentation about the /healthcheck/ endpoint (thanks Dan Carley)
  - improvements to router documentation pages (thanks Dan Carley)
  - fix code snippets in the services documentation page (thanks Leandro Souza)
  - typo and broken link fixes and structural improvements across all the documentation (thanks Dan Hilton).

Backward incompatible changes (action needed)

- As tsuru now creates containers per processes, whenever an application has more than one process, tsuru will forward requests to the process named "web". So, in a Procfile like the one below, "api" should be replaced with "web":

```
api: ./start-api
worker1: ./start-worker1
worker2: ./start-worker2
```

- You should change your fallback pool to default pool and to do that you can run a `tsuru-admin pool-update pool_name --default=true`
- tsr has been renamed to tsurud. Please update any procedures and workflows (including upstart and other init scripts).

10.1.5 tsr 0.11.3 release notes

Welcome to tsr 0.11.3!

tsr 0.11.3 includes fixes related to the deploy process:

- New configuration options related to timeouts in pub/sub redis connections. Default timeout values set so we can fail fast and not hang if there are connection problems accessing the redis server. See *config reference* for more details.
- Writing deploy execution logs is done in background to prevent slow storage backends from interfering in deploy time.
- Hitting Ctrl-C during a deploy does not stop the deploy process anymore. It can be followed again using `app-log`. #1238

10.1.6 tsr 0.11.2 release notes

Welcome to tsr 0.11.2!

tsr 0.11.2 includes some bug fixes and adds performance improvements related to the database management:

- Fix of database connection leaks across the entire code base, including a mechanism for automatically detecting new connection leaks. Also preventing new connection leaks by always closing the connection on object's finalizer.
- Fix compatibility with Docker 1.6+. Docker 1.6 introduced a new way of limiting container resources (CPU and memory). See issue #1213 for more details.
- Introduced a new configuration entry, for splitting the main database and the logs database, avoiding issues with global locks in MongoDB. For more details, see the *configuration docs*.
- Performance improvements in the log processing: properly ordering the logs and using less indexes to speed up write operations.
- Add a hard timeout to healthcheck requests, preventing stale of deployments while tsuru waits for the response of the application healthcheck. The current value for this timeout is 1 minute.

10.1.7 tsr 0.11.1 release notes

Welcome to tsr 0.11.1!

tsr 0.11.1 includes some bug fixes and adds profiling routes to enable further performance improvements to tsuru server:

- Remove support for round robin scheduler. Pools are mandatory since 0.11.0 and round robin didn't work anymore. This fix make this change clearer by validating tsuru.conf and explicitly preventing round robin scheduler from being used. Related to #1204
- Fix unit-remove from trying to remove a unit from nodes without units belonging to the specified application. Also making sure unit-remove choose the optimal node from which remove a unit (the one with the maximum number of unit from the same application). Related to #1204
- Updated monsterqueue version to avoid errors regarding unregistered tasks trying to be executed.
- Added HTTP routes to enable profiling tsuru server during its execution. This is intended to analyze and improve tsuru server performance under heavy loads.

10.1.8 tsr 0.11.0 release notes

Welcome to tsr 0.11.0!

These release notes cover the *new features*, *bug fixes*, *general improvements* and *backward incompatible changes* you'll want to be aware of when upgrading from tsr 0.10.0 or older versions.

Main new features

- Pool management overhaul. Now pools are a concept independent on the docker provisioner. You can have multiple pools associated with each team. If that's the case, when creating a new application, users will be able to choose which pool they want to use to deploy it.

  To support these features some client commands have changed, mainly `tsuru app-create` support a `--pool <poolname>` parameter.

Some action is needed to migrate old pool configuration to this new format. See *backward incompatible changes* section for more details. #1013

- Node auto scaling. It's now possible to enable automatic scaling of docker nodes, this will add or remove nodes according to rules specified in your tsuru.conf file. See node auto scaling topic and *config reference* for more details. #1110

Bug fixes

- Better handling erroneous tsuru.yaml files with tabs instead of spaces. #1165
- Restart after hooks now correctly run with environment variables associated to applications. #1159
- `tsuru app-shell` command now works with tsuru api under TLS. #1148
- Removing machines from IaaS succeed if referenced machine was already manually removed from IaaS. #1103
- Deploy details API call (/deploy/<id>) no longer fail with deploys originated by running `tsuru app-deploy`. #1098
- Cleaner syslog output without lots of apparmor entries. #997
- Running `tsuru app-deploy` on Windows now correctly handle directories and home path. #1168 #1169
- Application listing could temporarily fail after removing an application, this was fixed. #1176
- Running `tsuru app-shell` now correctly sets terminal size and TERM environment value, also container id is no longer ignored. #1112 #1114
- Fix bug in the flow of binding and unbinding applications to service instances. With this old bug, units could end-up being bound twice with a service instance.

Other improvements

- Limited number of goroutines started when initiating new units, avoiding starving docker with too many simultaneous connections. #1149
- There is now a tsr command to run necessary migrations when updating from older versions. You can run it with `tsr migrate` and it should not have side- effects on already up-to-date installations. #1137
- Added command `tsr gandalf-sync`, it should be called if Gandalf is activated on an existing tsuru api instance. It's responsible for copying existing users and teams credentials to Gandalf. Users added after Gandalf activation in tsuru.conf will already be created on Gandalf and this command doesn't needed to be called further. #1138
- It's now possible to remove all units from an application (thanks Lucas Weiblen). #1111.
- Removing units now uses the scheduler to correctly maintain units balanced across nodes when removing a number of units. #1109
- tsuru will keep trying to send image to registry during deploy for some time if the registry fails on the first request. #1099
- It's possible to use a docker registry with authentication support. See *config reference* for more details. #1182
- Partial support for docker distribution (registry 2.0). Image removal is not yet supported. #1175
- Improved *logging* support, allowing cloud admins to configure any of the three tsuru logging options: syslog, stderr or log file. At any time, it's possible to enable any of the three options.
- Running commands with `tsuru app-run` now log command's output to tsuru logs. #986
- Graceful shutdown of API when SIGTERM or SIGINT is received. The shutdown process now is:

  - Stop listening for new connections;
  - Wait for all ongoing connections to end;
  - Forcibly close `tsuru app-log -f` connections;
  - Wait for ongoing healing processes to end;
  - Wait for queue tasks to finish running;
  - Wait for ongoing auto scaling processes to end.

  #776

- Included lock information in API call returning application information. #1171
- Unit names now are prefixed with application's name (thanks Lucas Weiblen). #1160.
- Admin users can now specify which user they want removed. #1014
- It's now possible to change metadata associated with a node. #1016
- Users can now define a private environment variable with *tsuru env-set* (thanks Diogo Munaro).
- Better error messages on server startup when MongoDB isn't available (thanks Lucas Weiblen). #1125.
- Add timing information to the healthcheck endpoint, so tsuru admins can detect components that are slow, besides detecting which are down.
- Now *tsuru app-remove* does not guess app name (thanks Lucas Weiblen). #1106.
- General improvements in the documentation:
  - typo fixes and wording improvements to install and configuration pages (thanks Anna Shipman).
  - fix instructions for key management in the quickstart page (thanks Felippe Raposo).
  - improve documentation for the contributing page (thanks Lucas Weiblen).
  - fix user creation instruction in the installing page (thanks Samuel Roze).
  - fix wording and spelling in the Gandalf install page (thanks Martin Jackson).

Backward incompatible changes (action needed)

- There are two migrations that must run before deploying applications with tsr 0.11.0, they concern pools and can be run with `tsr migrate`. The way pools are handled has changed. Now it's possible for a team to have access to more than one pool, if that's the case the pool name will have to be specified during application creation. #1110
- Queue configuration is necessary for creating and removing machines using a IaaS provider. This can be simply done by indicating a MongoDB database configuration that will be used by tsuru for managing the queue. No external process is necessary. See *configuration reference* for more details. #1147
- Previously it was possible for more than one machine have the same address this could cause a number of inconsistencies when trying to remove said machine using `tsuru docker-node-remove --destroy`. To solve this problem tsuru will now raise an error if the IaaS provider return the same address of an already registered machine.

  If you already have multiple machines with the same address registered in tsuru, trying to add new machines will raise an error until the machines with duplicated address are removed.

10.1.9 tsr 0.10.2 release notes

Welcome to tsr 0.10.2!

tsr 0.10.2 includes one bug fixes to administration commands:

- tsuru-admin commands `container-move`, `containers-move` and `containers- rebalance` caused tsuru server to freeze. This issue was caused by a global mutex for all connections being permanently locked. This fix eliminates the global mutex and instead creates an independent lock per request. A performance improvement in api calls is also expected with this fix.

10.1.10 tsr 0.10.1 release notes

Welcome to tsr 0.10.1!

tsr 0.10.1 includes two improvements from the previous version and one bug fix:

- During start-up and image migration, skip applications that have already been moved (related to issue #712);
- Limit healing for Docker nodes. Now tsuru will heal Docker nodes when only there's a network error in the communication between the tsuru API and the Docker node with general operations, like pulling an image. When creating a container, any failure will count as a trigger for healing;
- Fix bug with authorization in the deploy hook, that allowed users to issue deployments to any application, via the API.

10.1.11 tsr 0.10.0 release notes

Welcome to tsr 0.10.0!

These release notes cover the *new features*, *bug fixes*, *backward incompatible changes* (specially the requirement on Gandalf and Docker versions), *general improvements* and *changes in the API* you'll want to be aware of when upgrading from tsr 0.9.0 or older versions.

What's new in tsr 0.10.0

- Now `tsuru app-run` and `tsuru-admin ssh` use `docker exec` to run commands on containers, this means that tsuru doesn't sshd inside containers anymore, making the containers more lightweight and saving some machine resources (issue #1002).
- It's now possible to have multiple routers configurations in your tsuru.conf file. The configuration to be used will be defined by which plan the application is using. See *routers* configuration reference and plan-create command for more details.

  For plans without a router configuration, the value defined in `docker:router` will still be used. So nothing will break with this change. See *docker:router* for more information.

  There's also a new router available: Galeb. For more details, please refer to *tsuru configuration reference* and Galeb's webpage.
- Users are now able to create apps with the same name used by a platform (issue #712).
- Extended the `healthcheck` entry in the `tsuru.yaml` file so users can specify a threshold of allowed failures. Please refer to the *tsuru.yaml documentation page* for more details (thanks Samuel ROZE).

- It's now possible to rollback your application to a previously deployed version. To support this feature the commands `app-deploy-list` and `app-deploy- rollback` were added. Also, all newly created application images in docker are versioned with `:vN`. You can change how many images will be available for rollback in `tsuru.conf`. See *config reference* and tsuru-client reference for more details.
- Gandalf is now optional. There's a new configuration entry for choosing the "repo-manager". For backwards compatibility purposes, when this entry is undefined, tsuru will use Gandalf. In order to disable Gandalf, users can set repo-manager to "none". When Gandalf is disabled, tsuru will not manage keys as well. For more details, see repository management page.
- New Ruby platform with support to multiple Ruby versions. Instead of having one platform per Ruby version, now users can just change the Ruby version they use by specifying it in the `Gemfile` or in the `.ruby-version` file.
- New PHP platform, with support to multiple PHP interpretors (FPM, mod_php) and frontends (Apache or nginx), including the support for configuring the virtual host (thanks Samuel ROZE).

Bug fixes

- Fix error message for unauthorized access in the `team-user-add` endpoint (issue #1006).
- Fix double restart bug on bind and unbind. When binding or unbinding apps, previous version of the tsuru-server daemon restarted the app twice, making the process _really_ slow when apps have a lot of units.
- Do not try to restart an app that has no units when removing environment variables.
- Bring back restart:after hooks, running them from the API after success in the healthcheck.

Other improvements in tsr 0.10.0

- tsuru doesn't store SSH public keys anymore, this handling is forwarded to the repository manager, and it's possible to run tsuru with no key management at all, by setting `repo-manager` to "none". Then the client will fail on `key-add`, `key-remove` and `key-list` with the message "key management is disabled" (issue #402).
- Improve user actions tracking. All app-related actions now use the `app=<appname>` format. Currently, these informations are available only in the database now, but in the future tsuru will expose all actions to admins, and may expose all actions of a user to themself.
- Support EBS optimized instances in the EC2 IaaS provider (issue #1058).
- Record the user that made the deploy when running `git push` (depends on upgrading the platforms and Gandalf).
- Improve user feedback (thanks Marc Abramowitz)
  - when the user creation fails
  - when failing to detect authentication scheme in the server
  - when making an unauthenticated requests, and receiving an unauthorized response
  - when resetting password
- Improve user feedback on API start-up (thanks Marc Abramowitz)
  - send fatal failures both to standard output and syslog (issue #1019)
  - properly report failure to connect to MongoDB
  - properly report failures to open the /etc/tsuru/tsuru.conf file

  - print the list of Docker nodes registered in the cluster
  - include more precise information about the router (including the configured domain and Redis endpoint, for Hipache)
- Properly set Content-Type headers in the API (thanks Marc Abramowitz)
- General improvements in the documentation:
  - Using rsyslog in tsuru applications (issue #796). See the logging documentation for more details;
  - Improvements in the recovery docs (thanks Mateus Del Bianco);
  - General grammar and RST syntax fixes in the documentation (thanks Alessandro Corbelli, Lucas Weiblen, Marc Abramowitz and Rogério Yokomizo);
  - Improve the contributing page;
  - Properly document the states of appplication units;
  - Split client documentation pages from the tsuru-server docs, there are now dedicated documentation sites for crane, tsuru-admin and tsuru-client;
  - Fix broken links in the documentation pages;
  - Improve Hipache installation docs;
  - Add documentation for the application metrics system (issue #990).
- Add instructions for upgrading Docker in the management documentation.

Backward incompatible changes

- This version of tsuru makes use of some features available only in the latest version of Gandalf, so if you plan to continue using Gandalf after this upgrade, you need to upgrade Gandalf to the version 0.6.0 (or bigger).
- This version of tsuru makes use of features available only from the 1.4 version of Docker, so before upgrading to tsuru-server 0.10.0, users must ensure that all Docker nodes are running Docker 1.4 or greater. Please refer to the upgrade Docker page for instructions on upgrading Docker with lesser downtime.
- tsuru changed the name of Docker images used for applications. During start-up, the server daemon will migrate images automatically. This may slow down the first start-up after the upgrade (issue #712).
- Drop support for Docker images that do not run tsuru-unit-agent. Starting at tsuru-server 0.10.0, every platform image must have tsuru-unit-agent installed, and ready to run.

API changes

tsuru-server 0.10.0 also include some changes in the API. Please refer to the API documentation page for more details.

- `/apps/{appname}/ssh`: new shell route to access app containers. In previous versions of API this route was in provision/docker package and just allowed admin access to app containers. Now, standart users and admin users can access app containers through ssh. Admins can access any app in tsuru and standart users can only access your apps.
- `/deploys`: allow non-admin users to issue requests to this endpoint. The response will list only deployments of applications that the user has access to. Admin users can still see all deployments from all applications (issue #1092).
- `/healthcheck`: tsuru now has an improved healthcheck endpoint, that will check the health of multiple components. In order to check everything, users should send a new request with the querystring parameter `check` set to `all`. Example: `GET /healthcheck?check=all` (issue #967).

- `/info`: this new endpoint returns meta information about the current running version of tsuru, like the server version and which components are enabled (issue #1093).
- `/services/instances/{instance}/{appname}`: bind and unbind endpoints now streams the progress of the binding/unbinding process (issue #963).
- `/tokens`: removed endpoint for generating an application token via the API. Users can no longer send POST requests to this URL.

10.1.12 tsr 0.9.1 release notes

Welcome to tsr 0.9.1!

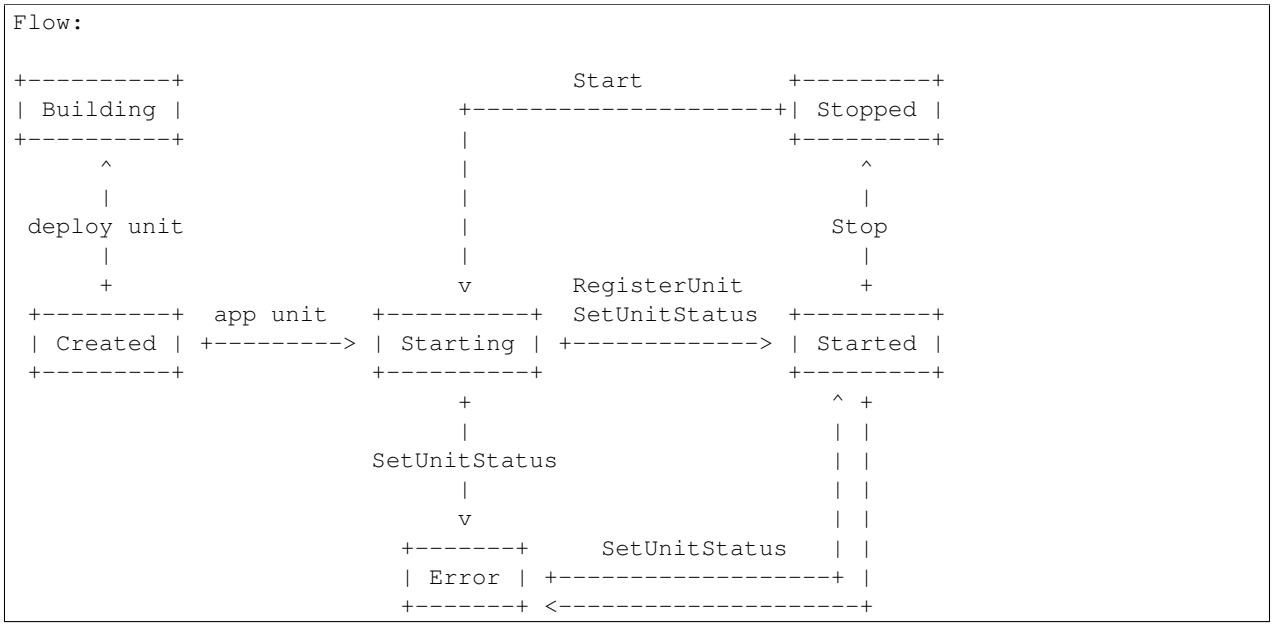
These release notes cover the *bug fixes*, *general improvements* and *changes in the API* you'll want to be aware of when upgrading from tsr 0.9.0 or older versions.

Bug fixes

- fix panic in the API when auto scale is enabled and the metric data is invalid.
- auto scale honors the min and max units when scaling
- `app-run` ignore build containers (issue #987).

Other improvements in tsr 0.9.1

- added some unit status and use correct status on build. Now the unit flow is:



API changes

- auto scale config info is now returned in the app-info endpoint.

10.1.13 tsr 0.9.0 release notes

Welcome to tsr 0.9.0!

These release notes cover the *new features*, *bug fixes*, *backward incompatible changes*, *general improvements* and *changes in the API* you'll want to be aware of when upgrading from tsr 0.8.0 or older versions.

What's new in tsr 0.9.0

- Now tsuru users can generate an API key, enabling authentication with no interactions required and having a token that never expires. Users can generate a new API key at any time using the command `tsuru token-regenerate` to replace the old one. To view the current key that you own, just use the command `tsuru token-show`.
- It's possible to use templates to create machines in the IaaS provider with `docker-node-add`. See machine-template-add command for more details.
- `TSURU_SERVICES` environment variable: this environment variable lists all service instances that the application is bound. This enables binding an application to multiple instances of a service (issue #991). For more details, check the TSURU_SERVICES documentation.
- auto scale: tsuru now includes an experimental support for auto scale. The auto scale uses the metric system to know when scale. To enable auto scale you should add the `autoscale: true` in then `tsuru.conf`.

Bug fixes

- app: SetEnvs not return error in apps with no units (issue #954).
- iaas/ec2: fixed panic after machine creation timeout.

Other improvements in tsr 0.9.0

- Improvements to EC2 IaaS provider, it now accepts user-data config through `iaas:ec2:user-data` and a timeout for machine creation with `iaas:ec2:wait-timeout` config.
- A new debug route is available in the API: `/debug/goroutines`. It can only be hit with admin credentials and will dump a trace of each running goroutine.

Backward incompatible changes

- Service API flow: the service API flow has changed, splitting the bind process in two steps: binding/unbinding the application and binding/unbinding the units. The old flow is now deprecated (issue #982).

API changes

For more details on the API, please refer to the tsuru API documentation.

- `/users/keys`: in previous versions of the API, this endpoint was used for adding and removing keys from the user account. Now it also lists the keys registered in the account of the user. Here is a summary of the behavior of this endpoint:
  - `GET`: return the list of keys registered in the user account
  - `POST`: add a new SSH key to the user account

  - `DELETE`: remove a SSH key from the user account

For the two last kind of requests, the user is now able to specify the name of the key, as well as the content.

10.1.14 tsr 0.8.2 release notes

Welcome to tsr 0.8.2!

These release notes cover the *bug fixes* you'll want to be aware of when upgrading from tsr 0.8.1 or older versions.

Bug fixes

- Requests to services using the proxy api call (/services/proxy/{instance}) now send the Host header of the original service endpoint. This allow proxied requests to be made to service apis running on tsuru. This fix is complementary to those made in proxy requests in 0.8.1.

10.1.15 tsr 0.8.1 release notes

Welcome to tsr 0.8.1!

These release notes cover the *bug fixes* you'll want to be aware of when upgrading from tsr 0.8.0 or older versions.

Bug fixes

- Fix trying to heal containers multiple times when it's unresponsive. Now tsuru will try to acquire a lock before storing the healing event. The healing will only be started if the lock has been successfully acquired and the container still exists in the database after the lock has been checked.
- Containers without exported ports (used during deploy) and with stopped state (set by running `tsuru stop` on the application) won't be healed anymore.
- The api call /services/proxy/{instance} route now will correctly handle HTTP headers. Previously, request headers weren't send from tsuru to the service, neither were response headers set by the service sent back to the client.

10.1.16 tsr 0.8.0 release notes

Welcome to tsr 0.8.0!

These release notes cover the *new features*, *bug fixes*, *backward incompatible changes*, *general improvements* and *changes in the API* you'll want to be aware of when upgrading from tsr 0.7.0 or older versions.

What's new in tsr 0.8.0

- tsuru now supports associating apps to plans which define how it can use machine resources, see *backward incompatible changes* for more information about which settings are no longer used with plans available, and how to use them.
- When using segregate scheduler, it's now possible to set a limit on how much memory of a memory will be reserved for app units. This can be done by defining some new config options. See the *config reference* for more details.

- The behavior of `restart`, `env-set` and `env-unset` has changed. Now they'll log their progress as they go through the following steps:
  - add new units;
  - wait for the health check if any is defined in tsuru.yaml;
  - add routes to new units;
  - remove routes from old units;
  - remove old units.
- tsuru now supports multiple configuration entries for the same IaaS provider, allowing a multi-region CloudStack or EC2 setup, for example. For more details, check the *Custom IaaS documentation*.

Bug fixes

- docker-pool-teams-add: fix to don't allow duplicate teams in a pool (issue #926).
- platform-remove: fix bug in the API that prevented the platform from being removed from the database (issue #936).
- Fix parameter mismatch between bind and unbind calls in service API (issue #794).

Other improvements in tsr 0.8.0

- Allow platform customization of environment for new units. This allow the use of virtualenv in the Python platform (contributes to fixing issue #928)
- Improve tsuru API acccess log (issue #608)
- Do not prevent users from running commands on units that are in the "error" state (issue #876)
- Now only the team that owns the application has access to it when the application is created. Other teams may be added in the future, using app-grant (issue #871)

Backward incompatible changes

The following config settings have been deprecated:

- docker:allow-memory-set
- docker:max-allowed-memory
- docker:max-allowed-swap
- docker:memory
- docker:swap

You should now create plans specifying the limits for memory, swap and cpu share. See tsuru-admin plan-create for more details.

API changes

For more details on the API, please refer to the tsuru API documentation.

- `/app/<appname>/run`: the endpoint for running commands has changed. Instead of streaming the output of the command in text format, now it streams it in JSON format, allowing clients to properly detect failures in the execution of the command.
- `/deploys`: list deployments in tsuru, with the possibility of filtering by application, service and/or user (issue #939).

10.1.17 tsr 0.7.2 release notes

Welcome to tsr 0.7.2!

These release notes cover the *bug fixes* you'll want to be aware of when upgrading from tsr 0.7.1 or older versions.

Bug fixes

- Fix bug which allow duplicated cname among apps;
- Fix bug on removing cname it doesn't exists;

10.1.18 tsr 0.7.1 release notes

Welcome to tsr 0.7.1!

These release notes cover the *bug fixes* you'll want to be aware of when upgrading from tsr 0.7.0 or older versions.

What's new in tsr 0.7.1

Bug fixes

- Fix bug causing deployment containers to be added in the router;
- Fix bug in deploy, causing it to run twice if tsuru_unit_agent is used and there's a failure during the deploy;

10.1.19 tsr 0.7.0 release notes

Welcome to tsr 0.7.0!

These release notes cover the *new features*, *bug fixes*, *backward incompatible changes* and *general improvements* you'll want to be aware of when upgrading from tsr 0.6.0 or older versions.

What's new in tsr 0.7.0

- quota management via API is back: now tsuru administrators are able to view and change the quota of a user of an application. It can be done from the remote API or using tsuru-admin (issue #869)
- deploy via upload: now it's possible to upload a tar archive to the API. In this case, users are able to just drop the file in the tsuru server, without using git. This feature enables the deployment of binaries, WAR files, and other things that may need local processing (issue #874). The tsuru client also includes a `tsuru deploy` command
- removing platforms via API: now tsuru administrators are able to remove platforms from tsuru. It can be done from the remote API or using tsuru-admin (issue #779)
- new apps now get a new environment variable: `TSURU_APPDIR`. This environment variable represents the path where the application was deployed, the root directory of the application (issue #783)

- now tsuru server will reload configuration on SIGHUP. Users running the API under upstart or other services like that are now able to call the `reload` command and get the expected behaviour (issue #898)
- multiple cnames: now it's possible to app have multiple cnames. The `tsuru set- cname` and `tsuru unset-cname` commands changed to `tsuru add-cname` and `tsuru remove-cname` respectively (issue #677).
- tsuru is now able to heal failing nodes and containers automatically, this is disabled by default. Instructions can be found in the *config reference*
- set app's owner team: now it's possible to user to change app's owner team. App's new owner team should be one of user's team. Admin user can change app's owner team to any team. (issue #894).
- Now it's possible to configure a health check request path to be called during the deployment process of an application. tsuru will make sure the health check is passing before switching the router to the newly created units. See *health check docs* for more details.

Bug fixes

- API: fix the endpoint for creating new services so it returns 409 Conflict instead of 500 when there's already a service registered with the provided name
- PlatformAdd: returns better error when an platform is added but theres no node to build the platform image (issue #906).

Other improvements in tsr 0.7.0

- API: improve the App swap endpoint, so it will refuse to swap incompatible apps. Two apps are incompatible if they don't use the same platform or don't have the same amount of units. Users can force the swap of incompatible apps by providing the force parameter (issue #582)
- API: admin users now see all service instances in the service instances list endpoint (issue #614)
- API: Handler that returns information about the deploy has implemented. Its included the diff attribute that returns the difference between the last commit and the preceding it.

Backward incompatible changes

- `tsr ssh-agent` has been totally removed, it's no longe possible to use it with tsuru server
- tsuru no longer accepts teams with space in the name (issue #674)
- tsuru no longer supports `docker:cluster:storage` set to `redis`, the only storage available is now `mongodb`. See *config reference* for more details. Also, there's a python script that can be used to migrate from `redis` to `mongodb`.
- Hooks semantic has changed, `restart:before-each` and `restart:after-each` no longer exist and now `restart:before` and `restart:afer` run on every unit. Also existing `app.yaml` file should be renamed to `tsuru.yaml`. See *hooks* for more details.
- Existing platform images should be updated due to changes in tsuru-circus and tsuru-unit-agent. Old platforms still work, but support will be dropped on the next version.
- router cnames should be migrate from string to list in redis. There is a script that can be used to migrate it.
- app should be migrate from string to list in mongo too. You can execute this code to do it:

```
db.apps.find().forEach(function(item) {
    cname = item.cname;
    item.cname !== "" ? item.cname = [cname]:item.cname = [];
    db.apps.save(item);
})
```

10.1.20 tsr 0.6.2 release notes

Welcome to tsr 0.6.2!

These release notes cover the *bug fixes* you'll want to be aware of when upgrading from tsr 0.6.1 or older versions.

What's new in tsr 0.6.2

Bug fixes

- Fix service proxy to read the request body properly.
- Fix deploy when trying to remove images from nodes.

10.1.21 tsr 0.6.1 release notes

Welcome to tsr 0.6.1!

These release notes cover the *bug fixes* you'll want to be aware of when upgrading from tsr 0.6.0 or older versions.

What's new in tsr 0.6.1

Bug fixes

- Fix eternal application locks after a Ctrl-C during deploy.
- Fix leak of connections to OAuth provider. Only users using auth:scheme as `oauth` are affected.
- Fix leak of connections to services.

10.1.22 tsr 0.6.0 release notes

Welcome to tsr 0.6.0!

These release notes cover the *new features*, *bug fixes* and *general improvements* you'll want to be aware of when upgrading from tsr 0.5.0 or older versions.

What's new in tsr 0.6.0

- Removed the ssh-agent dependency. Now tsuru will generate a RSA keypair per container, making it more secure and with one less agent running in the Docker hosts. Now a Docker host is just a host that runs Docker. tsuru server is still able to communicate with containers created using the ssh-agent, but won't create any new containers using a preconfigured SSH key. The version 0.7.0 will delete ssh-agent completely.

- tsuru now supports managing IaaS providers, this allow tsuru to provision new docker nodes making it a lot easier to install and maintain. The behavior of `docker-node-*` admin commands was changed to receive machine information and new commands have been added. See tsuru-admin for more details.

  Right now, EC2 and Cloudstack are supported as IaaS providers. You can see more details about how to configure them in the config reference

- Improved handling of unit statuses. Now the unit will communicate with the server, minute after minute, updating the status. This will work as a heart beat. So the unit will change to the status "error" whenever the heart beat fails after 4 minutes or the unit informs that the process failed to install.
- Add the capability to specify the owner of a service instance. tsuru will use this information when communicating with the service API
- During the deployment process, tsuru will now remove old units only after adding the new ones (related to the issue #511). It makes the process more stable and resilient.

Bug fixes

- fix security issue with user tokens: handlers that expected application token did not validate user access properly. With this failure, any authenticated user were able to add logs to an application, even if he/she doesn't have access to the app.

Breaking changes

- tsuru source no longer supports Go 1.1. It's possible that tsuru will build with Go 1.1, but it's no longer supported.
- tsuru_unit_agent package is not optional anymore, it must be available in the image otherwise the container won't start.
- docker cluster storage format in Redis has changed, also MongoDB is supported as an alternative to Redis. There is a migration script available which convert data in Redis to the new format, and also allows importing Redis data in MongoDB.
- since tsuru requires a service instance to have an owner team, i.e. a team that owns the service, users that are members of more than one team aren't able to create service instances using older versions of tsuru client (any version older than 0.11).
- in order to define the owner team of an already created service instance, tsuru administrators should run a migration script, that get's the first team of the service instance and use it as the owner team.
- all code related to beanstalkd has been removed, it isn't possible to use it anymore, users that were still using beanstalkd need to change the configuration of the API server to use redis instead

Other improvements

- improved documentation search and structure
- improved reliability of docker nodes, automatically trying another node in case of failures
- experimental support for automatically healing docker nodes added through the IaaS provider
- cmd: properly handle multiline cells in tables

10.1.23 tsr 0.5.3 release notes

Welcome to tsr 0.5.3!

These release notes cover the *bug fixes* you'll want to be aware of when upgrading from tsr 0.5.2 or older versions.

What's new in tsr 0.5.3

Bug fixes

- Fix leak of connections to Redis when using `queue: redis` in config.

10.1.24 tsr 0.5.2 release notes

Welcome to tsr 0.5.2!

These release notes cover the *new features* and *bug fixes* you'll want to be aware of when upgrading from tsr 0.5.1 or older versions.

What's new in tsr 0.5.2

Improvements

- improve the Docker cluster management so it keeps track of which node contains a certain image, so a request to remove an image from the cluster can be sent only to the proper nodes (docker-cluster #22).
- improve error handling on OAuth authentication

Bug fixes

- Check if node exists before excludind it (mongo doesn't return an error if I try to remove a node which not exists from a pool) (#840).
- Fix race condition in unit-remove that prevented the command from removing the requested number of units
- Fix app lock management in unit-remove

10.1.25 tsr 0.5.1 release notes

Welcome to tsr 0.5.1!

These release notes cover the *new features*, *bug fixes* and *backwards incompatible changes* you'll want to be aware of when upgrading from tsr 0.5.0 or older versions.

What's new in tsr 0.5.1

- **tsr api** now checks tsuru.conf file and refuse to start if it is misconfigured. It's also possible to exclusively test the config file with the -t flag. i.e.: running "tsr api -t". (#714).
- new command in the `tsuru-admin`: the command fix-containers will look for broken containers and fix their configuration within the router, and in the database

Bug fixes

- Do not lock application on `tsuru run`

Backwards incompatible changes

- **tsr collector** is no more. In the 0.5.0 release, collector got much less responsibilities, and now it does nothing, because it no longer exists. The last of its responsibilities is now available in the `tsuru-admin fix-containers` command.

10.1.26 tsr 0.5.0 release notes

Welcome to tsr 0.5.0!

These release notes cover the *new features* and *backwards incompatible changes* you'll want to be aware of when upgrading from tsr 0.4.0 or older versions.

What's new in tsr 0.5.0

Stability and Consistency

One of the main feature on this release is improve the stability and consitency of the tsuru API.

- prevent inconsitency caused by problems on deploy (#803) / (#804)
- units information is not updated by collector (#806)
- fixed log listener on multiple API hosts (#762)
- prevent inconsitency caused by simultaneous operations in an application (#789)
- prevent inconsitency cause by simultaneous `env-set` calls (#820)
- store information about errors and identify flawed application deployments (#816)

Buildpack

tsuru now supports deploying applications using Heroku Buildpacks.

Buildpacks are useful if you're interested in following Heroku's best practices for building applications or if you are deploying an application that already runs on Heroku.

tsuru uses Buildstep Docker image to deploy applications using buildpacks. For more information, take a look at the buildpacks documentation page.

Other features

- filter application logs by unit (#375)
- support for deployments with archives, which enables the use of the `pre-receive` Git hook, and also deployments without Git (#458, #442 and #701)
- stop and start commands (#606)
- oauth support (#752)

- platform update command (#780)
- support services with *https* endpoint (#812) / (#821)
- grouping nodes by pool in segregate scheduler. For more information you can see the docs about the segregate scheduler: Segregate Scheduler.

Platforms

- *deployment hooks* support for static and PHP applications (#607)
- new platform: buildpack (used for buildpack support)

Backwards incompatible changes

- Juju provisioner was removed. This provisioner was not being maintained. A possible idea is to use Juju in the future to provision the tsuru nodes instead of units
- ELB router was removed. This router was used only by juju.
- `tsr admin` was removed.
- The field `units` was removed from the collection `apps`. Information about units are now available in the provisioner. Now the unit state is controlled by provisioner. If you are upgrading tsuru from 0.4.0 or an older version you should run the MongoDB script below, where the *docker* collection name is the name configured by *docker:collection* in *tsuru.conf*:

```
var migration = function(doc) {
    doc.units.forEach(function(unit){
        db.docker.update({"id": unit.name}, {$set: {"status": unit.state}});
    });
};

db.apps.find().forEach(migration);
```

- The scheduler collection has changed to group nodes by pool. If you are using this scheduler you shoul run the MongoDB script below:

```
function idGenerator(id) {
    return id.replace(/\d+/g, "")
}

var migration = function(doc) {
    var id = idGenerator(doc._id);
    db.temp_scheduler_collection.update(
        {teams: doc.teams},
        {$push: {nodes: doc.address},
         $set: {teams: doc.teams, _id: id}},
        {upsert: true});
}
db.docker_scheduler.find().forEach(migration);
db.temp_scheduler_collection.renameCollection("docker_scheduler", true);
```

You can implement your own *idGenerator* to return the name for the new pools. In our case the *idGenerator* generates an id based on node name. It makes sense because we use the node name to identify a node group.

Features deprecated in 0.5.0

beanstalkd queue backend will be removed in 0.6.0.

10.1.27 tsr 0.4.0 release notes

Welcome to tsr 0.4.0!

These release notes cover the *new features* and *backwards incompatible changes* you'll want to be aware of when upgrading from tsr 0.3.x or older versions.

What's new in tsr 0.4.0

- redis queue backend was refactored.
- fixed output when service doesn't export environment variables (issue #772)

Docker

- refactored unit creation to be more atomic
- support for unit-agent (issue #633) - tsuru unit agent repository: https://github.com/tsuru/tsuru-unit-agent.
- added an administrative command to move and rebalance containers between nodes (issue #646). For more details, see the containers-rebalance reference.
- memory swap limit is configurable (issue #764)
- added a command to add a new platform (issue #780). For more details, see the platform-add reference.

Backwards incompatible changes

The S3 integration on app creation was removed. The config properties `bucket-support`, `aws:iam` `aws:s3` were removed as well.

You should use *tsuru* 0.9.0 and *tsuru-admin* 0.3.0 version.

10.1.28 tsr 0.3.12 release notes

Welcome to tsr 0.3.12!

These release notes cover the *new features* and *backwards incompatible changes* you'll want to be aware of when upgrading from tsr 0.3.11 or older versions.

What's new in tsr 0.3.12

Docker provisioner

- integrated the segregated scheduler with owner team - #753

Backwards incompatible changes

tsr 0.3.12 did not introduce any incompatible changes.

10.1.29 tsr 0.3.11 release notes

Welcome to tsr 0.3.11!

These release notes cover the *new features* and *backwards incompatible changes* you'll want to be aware of when upgrading from tsr 0.3.10 or older versions.

What's new in tsr 0.3.11

API

- Added app team owner - #619
- Expose public url in *create-app* - #724

Docker provisioner

- Add support to custom memory - #434

Backwards incompatible changes

All existing apps have no team owner. You can run the mongodb script below to automatically set the first existing team in the app as team owner.

```
db.apps.find({ teamowner: { $exists: false }}).forEach(
    function(app) {
        app.teamowner = app.teams[0];
        db.apps.save(app);
    }
);
```

10.1.30 tsr 0.3.10 release notes

Welcome to tsr 0.3.10!

These release notes cover the *new features* and *backwards incompatible changes* you'll want to be aware of when upgrading from tsr 0.3.9 or older versions.

What's new in tsr 0.3.10

API

- Improve feedback for duplicated users (issue #693)

Docker provisioner

- Update docker-cluster library, to fix the behavior of the default scheduler (issue #716)
- Improve debug logs for SSH (issue #665)
- Fix URL for listing containers by app

Backwards incompatible changes

tsr 0.3.10 did not introduce any incompatible changes.

10.1.31 tsr 0.3.9 release notes

Welcome to tsr 0.3.9!

These release notes cover the *new features* and *backwards incompatible changes* you'll want to be aware of when upgrading from tsr 0.3.8 or older versions.

What's new in tsr 0.3.9

API

- Login expose *is_admin* info.
- Changed get environs output data.

Backwards incompatible changes

tsr 0.3.9 has changed the API output data for get environs from an app.

You should use *tsuru* cli 0.8.10 version.

10.1.32 tsr 0.3.8 release notes

Welcome to tsr 0.3.8!

These release notes cover the *new features* and *backwards incompatible changes* you'll want to be aware of when upgrading from tsr 0.3.8 or older versions.

What's new in tsr 0.3.8

API

- Expose deploys of the app in the app-info API

Docker

- deploy hook support enviroment variables with space.

Backwards incompatible changes

tsr 0.3.7 does not introduce any incompatible changes.

10.1.33 tsr 0.3.7 release notes

Welcome to tsr 0.3.7!

These release notes cover the *new features* and *backwards incompatible changes* you'll want to be aware of when upgrading from tsr 0.3.6 or older versions.

What's new in tsr 0.3.7

API

- Improve administrative API for the Docker provisioner
- Store deploy metadata
- Improve healthcheck (ping MongoDB before marking the API is ok)
- Expose owner of the app in the app-info API

Backwards incompatible changes

tsr 0.3.7 does not introduce any incompatible changes.

10.1.34 tsr 0.3.6 release notes

Welcome to tsr 0.3.6!

These release notes cover the *new features* and *backwards incompatible changes* you'll want to be aware of when upgrading from tsr 0.3.5 or older versions.

What's new in tsr 0.3.6

Application state control

- Add new functionality to the API and provisoners: stop and starting an App

Services

- Add support for plans in services

Backwards incompatible changes

tsr 0.3.6 does not introduce any incompatible changes.

10.1.35 tsr 0.3.5 release notes

Welcome to tsr 0.3.5!

These release notes cover the *new features* and *backwards incompatible changes* you'll want to be aware of when upgrading from tsr 0.3.4 or older versions.

What's new in tsr 0.3.5

Bugfixes

- Fix administrative API for Docker provisioner

Backwards incompatible changes

tsr 0.3.5 does not introduce any incompatible changes.

10.1.36 tsr 0.3.4 release notes

Welcome to tsr 0.3.4!

These release notes cover the *new features* and *backwards incompatible changes* you'll want to be aware of when upgrading from tsr 0.3.3 or older versions.

What's new in tsr 0.3.4

Documentation improvements

- Improvements in the layout of the documentation

Bugfixes

- Swap address and cname on apps when running swap
- Always pull the image before creating the container

Backwards incompatible changes

tsr 0.3.4 does not introduce any incompatible changes.

10.1.37 tsr 0.3.3 release notes

Welcome to tsr 0.3.3!

These release notes cover the *new features* and *backwards incompatible changes* you'll want to be aware of when upgrading from tsr 0.3.2 or older versions.

What's new in tsr 0.3.3

Queue

- Add an option to use Redis instead of beanstalkd for work queue

In order to use Redis, you need to change the configuration file:

```
queue: redis
redis-queue:
  host: "localhost"
  port: 6379
  db: 4
  password: "your-password"
```

All settings are optional (`queue` will still default to "beanstalkd"), refer to configuration docs for more details.

Other improvements and bugfixes

- Do not depend on Docker code
- Improve the layout of the documentation
- Fix multiple data races in tests
- [BUGFIX] fix bug with unit-add and application image
- [BUGFIX] fix image replication on docker nodes

Backwards incompatible changes

tsr 0.3.3 does not introduce any incompatible changes.

10.1.38 tsr 0.3.2 release notes

Welcome to tsr 0.3.2!

These release notes cover the *new features* and *backwards incompatible changes* you'll want to be aware of when upgrading from tsr 0.3.1 or older versions.

What's new in tsr 0.3.2

Segregated scheduler

- Support more than one team per scheduler
- Fix the behavior of the segregated scheduler
- Improve documentation of the scheduler

API

- Improve administrative API registration

Other improvements and bugfixes

- Do not run restart on unit-add (nor unit-remove)
- Improve node management in the Docker provisioner
- Rebuild app image on every 10 deployment

Backwards incompatible changes

tsr 0.3.2 does not introduce any incompatible changes.

10.1.39 tsr 0.3.1 release notes

Welcome to tsr 0.3.0!

These release notes cover the *new features* and *backwards incompatible changes* you'll want to be aware of when upgrading from tsuru 0.3.0 or older versions.

What's new in tsr 0.3.1

Backwards incompatible changes

10.1.40 tsr 0.3.0 release notes

Welcome to tsr 0.3.0!

These release notes cover the *new features* and *backwards incompatible changes* you'll want to be aware of when upgrading from tsuru 0.2.x or older versions.

What's new in tsr 0.3.0

Support Docker 0.7.x and other improvements

- Fixed the 42 layers problem.
- Support all Docker storages.
- Pull image on creation if it does not exists.
- BUGFIX: when using segregatedScheduler, the provisioner fails to get the proper host address.
- BUGFIX: units losing access to services on deploy bug.

Improvements related to Services

- *bind* is atomic.
- *service-add* is atomic
- Service instance name is unique.
- Add support to bind an app without units.

Collector ticker time is configurable

Now you can define the collector ticker time. To do it just set on tsuru.conf:

```
collector:
    ticker-time: 120
```

The default value is 60 seconds.

Other improvements and bugfixes

- *unit-remove* does not block util all units are removed.
- BUGFIX: send on closed channel: https://github.com/tsuru/tsuru/issues/624.
- Api handler that returns information about all deploys.
- Refactored quota backend.
- New lisp platform. Thanks to Nick Ricketts.

Backwards incompatible changes

tsuru 0.3.0 handles quota in a brand new way. Users upgrading from 0.2.x need to run a migration script in the database. There are two scripts available: one for installations with quota enabled and other for installations without quota.

The easiest script is recommended for environments where quota is disabled, you'll need to run just a couple of commands in MongoDB:

```
% mongo tsuru
MongoDB shell version: x.x.x
connecting to: tsuru
> db.users.update({}, {$set: {quota: {limit: -1}}});
> db.apps.update({}, {$set: {quota: {limit: -1}}});
```

In environments where quota is enabled, the script is longer, but still simple:

```
db.quota.find().forEach(function(quota) {
    if(quota.owner.indexOf("@") > -1) {
        db.users.update({email: quota.owner}, {$set: {quota: {limit: quota.limit, inuse: quota.items
    } else {
        db.apps.update({name: quota.owner}, {$set: {quota: {limit: quota.limit, inuse: quota.items.le
    }
});
```

db.apps.update({quota: null}, {$set: {quota: {limit: -1}}}); db.users.update({quota: null}, {$set: {quota: {limit: -1}}}); db.quota.remove()

The best way to run it is saving it to a file and invoke MongoDB with the file parameter:

```
% mongo tsuru <filename.js>
```

10.2 tsuru

tsuru is the tsuru client. For details on releases of the client, check the release history in the tsuru-client repository at GitHub.

10.3 tsuru-admin

tsuru-admin is the tsuru administrative client. For details on releases of tsuru-admin, check the release history in the tsuru-admin repository at GitHub.

10.4 crane

crane is the command line interface used by service providers. For details on releases of crane, check the release history in the crane repository at GitHub.

CHAPTER 11

Roadmap

11.1 Release Process

We use GitHub's milestones to releases' planning and anyone is free to suggest an issue to a milestone, and discuss about any issue in the next tsuru version. We also have internal goals as listed bellow and our focus will be these goals. But it's not immutable, we can change any goal at any time as community need.

At globo.com we have goals by quarter of a year (short term goals bellow), but it doesn't mean that there's only one release per quarter. Our releases have one or more main issues and minor issues which can be minor bugfixes, ground work issue and other "not so important but needed" issues.

You can suggest any issue to any milestones at any time, and we'll discuss it in the issue or in Gitter.

11.2 Next Release 0.12.0

- Lean containers (issue #1136)
- Dockerize tsuru installation (issue #1091)

11.3 Long term Goals

These are our goals to 1.0 version.

- review platform management.

  We are thinking to change our way to manage platform. Today tsuru has its own platform. But we have a lot of problems to mantain it. In other way, we have buildpacks and we can use it to provide any platform we want, but there's no free lunch.' Buildpack can be built by anyone and are updated "automagically", so your application may stop to deploy properly, totally out of the blue.

- docker images with envs.
- get logs and metrics from outside of app container.
- improve *app-swap*
- improve plugins