tsuru Documentation

Release 0.8.0

timeredbull

November 04, 2014

Contents

1 Understanding **3**
- 1.1 Overview . . . 3
- 1.2 Concepts . . . 4
- 1.3 Architecture . . . 5

2 Installing **7**
- 2.1 Gandalf . . . 7
- 2.2 API Server . . . 8
- 2.3 Hipache Router . . . 10
- 2.4 Adding Nodes . . . 11

3 Managing **13**
- 3.1 Installing platforms . . . 13
- 3.2 Creating a platform . . . 13
- 3.3 Backing up tsuru database . . . 14
- 3.4 Segregate Scheduler . . . 15

4 Using **17**
- 4.1 Installing tsuru clients . . . 17
- 4.2 Building your app in tsuru . . . 18
- 4.3 Deploying Python applications in tsuru . . . 19
- 4.4 Deploying Ruby applications in tsuru . . . 28
- 4.5 Deploying Go applications in tsuru . . . 35
- 4.6 Deploying PHP applications in tsuru . . . 38
- 4.7 Using Buildpacks . . . 44
- 4.8 Recovering an application . . . 45
- 4.9 Procfile . . . 46
- 4.10 tsuru.yaml . . . 46
- 4.11 unit states . . . 47
- 4.12 Guide to create tsuru cli plugins . . . 48
- 4.13 Application Deployment . . . 49

5 Contributing **51**
- 5.1 Development environment . . . 51
- 5.2 Running the tests . . . 51
- 5.3 Writing docs . . . 51
- 5.4 Building docs . . . 51
- 5.5 Community . . . 52

5.6 Release Process . . . 52

6 Services **55**
6.1 Crane usage . . . 55
6.2 API workflow . . . 56
6.3 Building your service . . . 60

7 Reference **67**
7.1 Configuring tsuru . . . 67
7.2 API reference . . . 78
7.3 Services . . . 91
7.4 tsuru-admin usage . . . 91
7.5 Client usage . . . 94

8 Frequently Asked Questions **97**
8.1 How does environment variables work? . . . 97
8.2 How does the quota system works? . . . 97
8.3 How routing works? . . . 97
8.4 How are Git repositories managed? . . . 98
8.5 Client installation fails with “undefined: bufio.Scanner”. What does it mean? . . . 98

9 Release notes **99**
9.1 tsr . . . 99
9.2 tsuru . . . 116
9.3 tsuru-admin . . . 117
9.4 crane . . . 117

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 the 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. You can write apps in the programming language of your choice, back it with add-on resources such as SQL and NoSQL databases, memcached, redis, and many others. You manage your app using the tsuru command-line tool and you deploy code using the Git revision control system, all running on the tsuru infrastructure.

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`, tsuru builds a Docker image and then distributes it as *units* 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.

1.2.4 Applications

An application consists of:

- the program's source code - e.g.: Python, Ruby, Go, PHP
- 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

API component is a RESTful API server written with `Go`. The API is responsible to the deploy workflow and 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, users and provide 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.

CHAPTER 2

Installing

If you're want to try tsuru with a minimum effort we recommend you to use tsuru Now (or tsuru-bootstrap, that runs tsuru Now on vagrant).

tsuru Now will install tsuru API, tsuru Client, tsuru Admin, and all its dependencies and on a single machine. Including the docker node which will run deployed applications.

However, 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 a 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 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 server archives
- archive-server: uses tsuru's archive-server to store and serve archives

- swift: uses Swift to store and server 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-se
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 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* were 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
queue: redis
redis-queue:
    host: <your-redis-server>
    port: 6379
git:
    unit-repo: /home/application/current
    api-server: http://<your-gandalf-server>:8000
provisioner: docker
docker:
    segregate: false
    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
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-admin` 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-admin

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

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

$ tsuru-admin team-create admin
```

And that's it, you now have registered an 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*, 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:

```
$ tsr 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 Installing

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

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

2.3.2 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 a physical or virtual machines with a Docker installation.

Nodes can be either 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 automatically managed option is preferred starting with tsuru 0.6.0. There are advantages like automatically healing and scaling of Nodes which will be implemented in the future.

The sections below describe how to add managed nodes and manually created nodes respectively.

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, this will be something like:

```
iaas:
  default: ec2
  node-protocol: http
  node-port: 2375
  ec2:
    key-id: xxxxxxxxxx
    secret-key: yyyyyyyyyyyy
```

After you have everything configured, adding a new docker done 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 metatada to the Node and you can send After registering your node, you can list it calling *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 Manually created nodes

To add a previously provisioned nodes you call the *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
```

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-certific
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

If you create a platform using docker, you can use the tsuru-admin cmd to add that.

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

3.3 Backing up tsuru database

In the tsuru repository, you will find two useful scripts in the directory `misc/mongodb`: `backup.bash` and `healer.bash`. In this page you will learn the purpose of these scripts and how to use them.

3.3.1 Dependencies

The script `backup.bash` uses S3 to store archives, and `healer.bash` downloads archives from S3 buckets. In order to communicate with S3 API, both scripts use s3cmd.

So, before running those scripts, make sure you have installed s3cmd. You can install it using your preferred package manager. For more details, refer to its download documentation.

After installing s3cmd, you will need to configure it, by running the command:

```
$ s3cmd --configure
```

3.3.2 Saving data

The script `backup.bash` runs `mongodump`, creates a tar archive and send the archive to S3. Here is how you use it:

```
$ ./misc/mongodb/backup.bash s3://mybucket localhost database
```

The first parameter is the S3 bucket. The second parameter is the database host. You can provide just the hostname, or the host:port (for example, 127.0.0.1:27018). The third parameter is the name of the database.

3.3.3 Automatically restoring on data loss

The other script in the `misc/mongodb` directory is `healer.bash`. This script checks a list of collections and if any of them is gone, download the last three backup archives and fix all gone collections.

This is how you should use it:

```
$ ./misc/mongodb/healer.bash s3://mybucket localhost mongodb repositories users
```

The first three parameters mean the same as in the backup script. From the fourth parameter onwards, you should list the collections. In the example above, we provided two collections: "repositories" and "users".

3.4 Segregate Scheduler

3.4.1 Overview

tsuru uses schedulers to chooses which node an unit should be deployed. There are two schedulers: *round robin* and *segregate scheduler*.

The default one is *round robin*, this page describes what the *segregate scheduler* does and how to enable it.

3.4.2 How it works

Segregate scheduler is a scheduler that segregates the units between nodes by team.

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 with the pool metadata information associated to the team owning the application being deployed.

Configuration and setup

To use the *segregate scheduler* you need to enable the segregate mode in `tsuru.conf`:

```
docker:
  segregate: true
```

Adding a pool

Using *tsuru-admin* you create a pool:

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

Adding teams to a pool

You can add one or more teams at once.

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

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

Listing a pool

To list pools you do:

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

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
```

Removing a pool

To remove a pool you do:

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

Removing teams from a pool

You can remove one or more teams at once.

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

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

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. For more detail, check *crane usage*;
- **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.

- Downloading binaries (Mac OS X and Linux)
- Using homebrew (Mac OS X only)
- Using the PPA (Ubuntu only)
- Using AUR (ArchLinux only)
- Build from source (Linux and Mac OS X)

4.1.1 Downloading binaries (Mac OS X and Linux)

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 requires Go 1.2 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 Using AUR (ArchLinux only)

Archlinux users can build and install tsuru client from AUR repository, Is needed to have installed yaourt program.

You can run:

```
$ yaourt -S tsuru
```

4.1.5 Build from source (Linux and Mac OS X)

Note: If you're feeling adventurous, you can try it on other systems, like FreeBSD, OpenBSD or even Windows. 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 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 user-create command:

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

`user-create` will ask for your password twice.

4.2.3 Login

To login in tsuru, you use the login command, you will be asked for your password:

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

4.2.4 Deploy an application

Choose from the following getting started tutorials to learn how to deploy your first application using a supported language or framework:

- *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 app-create command:

```
$ 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 platform-list command.

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

```
$ tsuru app-list
+-------------+-------------------------+---------+--------+
| Application | Units State Summary     | Address | Ready? |
+-------------+-------------------------+---------+--------+
| blog        | 0 of 0 units in-service |         | No     |
+-------------+-------------------------+---------+--------+
```

Once your app is ready, you will be able to deploy your code, e.g.:

```
$ tsuru app-list
+-------------+-------------------------+-------------+--------+
| Application | Units State Summary     | Address     | Ready? |
+-------------+-------------------------+-------------+--------+
| blog        | 0 of 1 units in-service |             | Yes    |
+-------------+-------------------------+-------------+--------+
```

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 app-info command:

```
$ tsuru app-info --app blog
Application: blog
Repository: git@git.tsuru.io:blog.git
Platform: python
```

```
Teams: tsuruteam
Address:
```

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@git.tsuru.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@git.tsuru.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 key-add command:

```
$ tsuru key-add ~/.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@git.tsuru.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@git.tsuru.io:blog.git
Platform: python
Teams: tsuruteam
Address: blog.cloud.tsuru.io
Units:
+--------------+---------+
| Unit         | State   |
+--------------+---------+
| 9e70748f4f25 | started |
+--------------+---------+
```

For more details on the `--app` flag, see "Guessing app names" section of tsuru command documentation.

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 bellow:

```
% 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@git.tsuru.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@git.tsuru.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@git.tsuru.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            | Ready? |
+-------------+-------------------------+--------------------+--------+
| blog        | 1 of 1 units in-service | blog.cloud.tsuru.io | Yes    |
+-------------+-------------------------+--------------------+--------+
```

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 service-list command:

```
$ 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 service-add command:

```
$ 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 bind command:

```
$ tsuru 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@git.tsuru.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 run command to execute commands in the machine, so for running `syncdb` we could write:

```
$ tsuru 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@git.tsuru.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 app-create command:

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

For Ruby, the app platform is, guess what, `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 platform-list command.

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

```
$ tsuru app-list
+-------------+------------------------+---------+--------+
| Application | Units State Summary    | Address | Ready? |
+-------------+------------------------+---------+--------+
```

```
| blog        | 0 of 0 units in-service |         | No     |
+-------------+-------------------------+---------+--------+
```

Once your app is ready, you will be able to deploy your code, e.g.:

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

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 app-info command:

```
$ tsuru app-info --app blog
Application: blog
Repository: git@cloud.tsuru.io:blog.git
Platform: ruby
Teams: tsuruteam
Address:
```

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@cloud.tsuru.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@cloud.tsuru.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 key-add command:

```
$ tsuru key-add ~/.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@cloud.tsuru.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@cloud.tsuru.io:blog.git
Platform: ruby
Teams: tsuruteam
Address: blog.cloud.tsuru.io
Units:
+--------------+---------+
| Unit         | State   |
+--------------+---------+
| 9e70748f4f25 | started |
+--------------+---------+
```

For more details on the `--app` flag, see "Guessing app names" section of tsuru command documentation.

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 bellow:

```
$ 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@cloud.tsuru.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@cloud.tsuru.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            | Ready? |
+-------------+-------------------------+--------------------+--------+
```

```
| blog        | 1 of 1 units in-service | blog.cloud.tsuru.io | Yes    |
+-------------+-------------------------+---------------------+--------+
```

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 service-list command:

```
$ 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 service-add command:

```
$ 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 bind command:

```
$ tsuru 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@cloud.tsuru.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 run command to execute commands in the machine, so for running `rake db:migrate` we could write:

```
$ tsuru 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@cloud.tsuru.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@cloud.tsuru.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 http://blog.cloud.tsuru.io/posts/.

4.4.9 Going further

For more information, you can dig into tsuru docs, or read complete instructions of use for 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 app-create command:

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

For go, the app platform is, guess what, `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 platform-list command.

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

```
$ tsuru app-list
+-------------+-------------------------+---------+--------+
| Application | Units State Summary     | Address | Ready? |
+-------------+-------------------------+---------+--------+
| helloworld  | 0 of 0 units in-service |         | No     |
+-------------+-------------------------+---------+--------+
```

Once your app is ready, you will be able to deploy your code, e.g.:

```
$ tsuru app-list
+-------------+-------------------------+-------------+--------+
| Application | Units State Summary     | Address     | Ready? |
+-------------+-------------------------+-------------+--------+
| helloworld  | 0 of 0 units in-service |             | Yes    |
+-------------+-------------------------+-------------+--------+
```

4.5.3 Application code

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

```
package main

import (
    "fmt"
    "net/http"
)

func main() {
    http.HandleFunc("/", hello)
    fmt.Println("listening...")
    err := http.ListenAndServe(":8888", 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 app-info command:

```
$ tsuru app-info --app blog
Application: go
Repository: git@cloud.tsuru.io:blog.git
Platform: go
Teams: myteam
Address:
```

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@cloud.tsuru.io:helloworld.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 /
####################################
#          OMIT (see below)        #
####################################
remote:  ---> App will be restarted, please check its log for more details...
remote:
To git@cloud.tsuru.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 key-add command:

```
$ tsuru key-add ~/.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@cloud.tsuru.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@cloud.tsuru.io:blog.git
Platform: go
Teams: myteam
Address: helloworld.cloud.tsuru.io
Units:
+--------------+---------+
| Unit         | State   |
+--------------+---------+
| 9e70748f4f25 | started |
+--------------+---------+
```

For more details on the `--app` flag, see "Guessing app names" section of tsuru command documentation.

4.5.5 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: ./main
```

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@cloud.tsuru.io:helloworld.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             | Ready? |
+-------------+-------------------------+---------------------+--------+
| helloworld  | 1 of 1 units in-service | blog.cloud.tsuru.io | Yes    |
+-------------+-------------------------+---------------------+--------+
```

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

Now we can access your *app* in the URL http://helloworld.cloud.tsuru.io/.

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 PHP applications in tsuru

4.6.1 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.

4.6.2 Creating the app within tsuru

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

```
$ 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 platform-list command.

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

```
$ tsuru app-list
+-------------+-------------------------+---------+--------+
| Application | Units State Summary     | Address | Ready? |
+-------------+-------------------------+---------+--------+
| blog        | 0 of 0 units in-service |         | No     |
+-------------+-------------------------+---------+--------+
```

Once your app is ready, you will be able to deploy your code, e.g.:

```
$ tsuru app-list
+-------------+-------------------------+-------------+--------+
| Application | Units State Summary     | Address     | Ready? |
+-------------+-------------------------+-------------+--------+
| blog        | 0 of 1 units in-service |             | Yes    |
+-------------+-------------------------+-------------+--------+
```

4.6.3 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'
```

4.6.4 Git deployment

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

```
$ tsuru app-info --app blog
Application: blog
Repository: git@git.tsuru.io:blog.git
Platform: php
Teams: tsuruteam
Address:
```

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@git.tsuru.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@git.tsuru.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 key-add command:

```
$ tsuru key-add ~/.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@git.tsuru.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@git.tsuru.io:blog.git
Platform: php
Teams: tsuruteam
Address: blog.cloud.tsuru.io
Units:
+--------------+---------+
| Unit         | State   |
+--------------+---------+
| 9e70748f4f25 | started |
+--------------+---------+
```

For more details on the `--app` flag, see "Guessing app names" section of tsuru command documentation.

4.6.5 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 bellow:

```
% 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@git.tsuru.io:ingress.git
 * [new branch]      master -> master
```

4.6.6 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             | Ready? |
+-------------+-------------------------+---------------------+--------+
| blog        | 1 of 1 units in-service | blog.cloud.tsuru.io | Yes    |
+-------------+-------------------------+---------------------+--------+
```

4.6.7 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 service-list command:

```
$ tsuru service-list
+----------------+-----------+
| Services       | Instances |
+----------------+-----------+
| mongodb        |           |
| 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 service-add command:

```
$ 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 bind command:

```
$ tsuru 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 bellow, 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 bellow:

```
$ 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 bellow:

```
$ 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.

4.6.8 Going further

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

4.7 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.

4.7.1 Creating an Application

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

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

4.7.2 Deploying your Application

Use git push master to deploy your application.

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

4.7.3 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.

4.7.4 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.

4.7.5 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.

4.8 Recovering an application

Your application may be downtime for a number of reasons. This page will help you discover why and what you can do to fix the problem.

4.8.1 Check your application logs

The first step is to check the application logs. To view your logs, run:

```
$ tsuru log -a appname
```

4.8.2 Restart your application

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

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

4.8.3 Checking units status

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

4.9 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
```

4.9.1 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 bellow:

```
<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.

4.9.2 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 *Procfile*: http://honcho.rtfd.org/en/latest/using_procfiles.html.

4.10 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`. (`app.yaml` or `app.yml` are also supported for backward compatibility reasons, however this will be dropped soon.)

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.

4.10.1 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.

4.10.2 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.*
```

- `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).

4.11 unit states

4.11.1 pending

Is when the unit is waiting to be provisioned by the tsuru provisioner.

4.11.2 bulding

Is while the unit is provisioned, it's occurs while a deploy.

4.11.3 error

Is when the an error occurs caused by the application code.

4.11.4 down

Is when an error occurs caused by tsuru internal problems.

4.11.5 unreachable

Is when the app process is up, but it is not bound to the right host ("0.0.0.0") and/or right port ($PORT). If your process is a worker it's state will be *unreachable*.

4.11.6 started

Is when the app process is up binded in the right host ("0.0.0.0") and right port ($PORT).

4.12 Guide to create tsuru cli plugins

4.12.1 Installing a plugin

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

`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>
```

4.12.2 Listing installed plugins

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

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

4.12.3 Executing a plugin

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

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

4.12.4 Removing a plugin

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

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

4.12.5 Creating your own plugin

Everything you need to do is to create a new file that can be executed. You can use Bash, Python, Ruby, eg.

Let's create a Hello world plugin that prints "hello world" as output. Let's use `bash` to write our new plugin.

```
#! /bin/bash
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.

4.13 Application Deployment

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

4.13.1 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": "",
        "TEST_NAME": "test",
    }
}
```

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>.

CHAPTER 5

Contributing

- Source hosted at GitHub
- Report issues on GitHub Issues

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

5.1 Development environment

See this guide to *to setup a development environment using vagrant*.

And follow our *coding style guide*.

5.2 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
```

5.3 Writing docs

tsuru documentation is written using Sphinx, which uses RST. Check these tools docs to learn how to write docs for tsuru.

5.4 Building docs

In order to build the HTML docs, just run on terminal:

```
$ make doc
```

5.5 Community

5.5.1 irc channel

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

5.6 Release Process

Tsuru major releases are guided by *github milestones<https://github.com/tsuru/tsuru/milestones/>_*. New releases should be generated by *make release version=new-version-number*.

5.6.1 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 by the <Something> *constructor*:

```
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 collection *method* that adds one or more elements:

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

Create<Something>

it's a *function* that's save an instance of <Something> in the database. Should receives an instance of <Something>.

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

Delete<Something>

it's a *function* that's delete an instance of <Something> from database.

Remove<Something>

it's opposite of Add<Something>.

5.6.2 Building a development environment with Vagrant

First, make sure that virtualbox, 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*. It will take a time:

```
cd tsuru-bootstrap
vagrant up
```

After it, configure the tsuru target with the address of the server that's running by vagrant:

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

Now you can create your user and deploy your apps.

CHAPTER 6

Services

6.1 Crane usage

First, you must set the target with your server url, like:

```
$ crane target tsuru.myhost.com
```

After that, all you need is to create a user and authenticate:

```
$ crane user-create youremail@gmail.com
$ crane login youremail@gmail.com
```

To generate a service 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
  test: test-endpoint.com:8080
```

The manifest.yaml is used by crane to define an id and an endpoint to your service.

To submit your new service, you can run:

```
$ crane create path/to/your/manifest.yaml
```

To list your services:

```
$ crane list
```

This will return something like:

```
+----------+-----------+
| Services | Instances |
+----------+-----------+
| mysql    | my_db     |
+----------+-----------+
```

To update a service manifest:

```
$ crane update path/to/your/manifest.yaml
```

To remove a service:

```
$ crane remove service_name
```

It would be nice if your service had some documentation. To add a documentation to you service you can use:

```
$ crane doc-add service_name path/to/your/docfile
```

Crane will read the content of the file and save it.

To show the current documentation of your service:

```
$ crane doc-get service_name
```

6.1.1 Further instructions

For a complete reference, check the documentation for crane command: http://godoc.org/github.com/tsuru/crane.

6.2 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 bind`)
- unbind an app from the service instance (`tsuru 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`)

6.2.1 Authentication

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

6.2.2 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
```

6.2.3 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.

6.2.4 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: 19
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
```

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.

6.2.5 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 bind mysql_instance --app my_app
```

tsuru calls the service API to bind an app with an instance via POST on `/resources/<service-instance-name>/bind` (please notice that tsuru does not include a trailing slash) 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 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.

6.2.6 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 unbind mysql_instance --app my_app
```

tsuru calls the service API to unbind the app from the instance via DELETE on `/resources/<service-instance-name>/bind` (please notice that tsuru does not include a trailing slash). 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 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.

6.2.7 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.

6.2.8 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.

6.2.9 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"}]
```

6.3 Building your service

6.3.1 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.

6.3.2 Creating your service API

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

6.3.3 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` 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", methods=["POST"])
def bind(name):
    app_host = request.form.get("app-host")
    unit_host = request.form.get("unit-host")
    # use name, app_host and unit_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`.

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", methods=["DELETE"])
def unbind(name, host):
    app_host = request.form.get("app-host")
    unit_host = request.form.get("unit-host")
    # use name, app-host and unit-host to remove the bind
    return "", 200
```

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

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


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

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

6.3.4 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: fakeserviceid1
password: secret123
endpoint:
    production: fakeserviceid1.com
```

submit your service: Submiting your service API

6.3.5 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*.

CHAPTER 7

Reference

7.1 Configuring tsuru

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

7.1.1 Notation

tsuru uses a colon to represent nesting in YAML. So, whenever this document say 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>
```

7.1.2 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.

Database access

tsuru uses MongoDB as database manager, to store information about users, VM's, and its components. Regarding database control, you're able to define to which database server tsuru will connect (providing a MongoDB connection string). The database related options are listed below:

database:url

`database:url` is the database connection string. It is a mandatory setting and has no default value. Examples of strings include the basic "127.0.0.1" and the 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".

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 would not be able to reset their password automatically.

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.

Git configuration

tsuru uses Gandalf to manage git repositories. Gandalf exposes a REST API for repositories management, and tsuru uses it. So tsuru requires information about the Gandalf HTTP server, and also its git-daemon and SSH service.

tsuru also needs to know where the git repository will be cloned and stored in units storage. Here are all options related to git repositories:

git:unit-repo

`git:unit-repo` is the path where tsuru will clone and manage the git repository in all units of an application. This is where the code of the applications will be stored in their units. Example of value: `/home/application/current`.

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`.

git:rw-host

`git:rw-host` is the host that will be used to build the push URL. For example, when the value is "tsuruhost.com", the push URL will be something like git@tsuruhost.com:<app-name>.git.

git:ro-host

`git:ro-host` is the host that units will use to clone code from users applications. It's used to build the read only URL of the repository. For example, when the value is "tsuruhost.com", the read-only URL will be something like git://tsuruhost.com/<app-name>.git.

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 and 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, the user creation URL is not registered in the server.

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.

Currently, tsuru supports only `redis` as queue backend. Creating a new queue provider is as easy as implementing an interface.

queue

`queue` is the name of the queue implementation that tsuru will use. This setting defaults to `redis`.

redis-queue:host

`redis-queue:host` is the host of the Redis server to be used for the working queue. This settings is optional and defaults to "localhost".

redis-queue:port

`redis-queue:port` is the port of the Redis server to be used for the working queue. This settings is optional and defaults to 6379.

redis-queue:password

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

redis-queue:db

`redis-queue:db` is the database number of the Redis server to be used for the working queue. This settings is optional and defaults to 3.

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".

Log

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. By default tsuru logs to syslog. If this is set, make sure tsuru has permissions to write to this file

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.

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>`

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:registry

For tsuru to work with multiple docker nodes, you will need a docker-registry. This should be in the form of `hostname:port`.

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:router

Router to be used to distribute requests to units. Right now only `hipache` is supported.

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:segregate

Enable segregate scheduler. See *Segregate Scheduler* for details.

docker:scheduler:total-memory-metadata

Only valid if `docker:segregate` is true. 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

Only valid if `docker:segregate` is true. 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 only allow the creation of new units if there is at least one server with enough unreserved memory to fit the amount of memory needed by the unit, based on which plan was used to create the application.

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:ssh:add-key-cmd

The command that will be called with the ssh public key created for the application. This allows us to connect directly to a running container using ssh. The value expected for basebuilder platforms is `/var/lib/tsuru/add-key`.

docker:ssh:public-key

Deprecated. You shouldn't set this value anymore.

docker:ssh:user

The user used to connect via ssh to running containers. 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.

7.1.3 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.

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.

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.

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 bellow 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
```

7.1.4 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
queue: redis
redis-queue:
    host: <your-redis-server>
    port: 6379
git:
    unit-repo: /home/application/current
    api-server: http://<your-gandalf-server>:8000
provisioner: docker
docker:
    segregate: false
    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
hipache:
    domain: <your-hipache-server-ip>.xip.io
    redis-server: <your-redis-server-with-port>
```

7.2 API reference

7.2.1 1. Endpoints

1.1 Apps

List apps

- Method: GET
- URI: /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

- URI: /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
- URI: /apps/:appname

Returns 200 in case of success.

Example:

```
DELETE /apps/myapp HTTP/1.1
```

Create an app

- Method: POST
- URI: /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.plataformas.glb.com:ble.git"}
```

Restart an app

- Method: GET
- URI: /apps/<appname>/restart

Returns 200 in case of success.

Example:

```
GET /apps/myapp/restart HTTP/1.1
```

Get app environment variables

- Method: GET
- URI: /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
- URI: /apps/<appname>/env

Returns 200 in case of success.

Example:

```
POST /apps/myapp/env HTTP/1.1
```

Delete an app environment

- Method: DELETE
- URI: /apps/<appname>/env

Returns 200 in case of success.

Example:

```
DELETE /apps/myapp/env HTTP/1.1
```

Swapping two apps

- Method: PUT
- URI: /swap?app1=appname&app2=anotherapp

Returns 200 in case of success.

Example:

```
PUT /swap?app1=myapp&app2=anotherapp
```

Get app log

- Method: GET
- URI: /apps/:appname/logs?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/logs?lines=1&source=web&unit=83535b503c96
Content-Length: 142
[{"Date":"2014-09-26T00:26:30.036Z","Message":"Booting worker with pid: 53","Source":"web","AppName"
```

1.2 Services

List services

- Method: GET
- URI: /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
- URI: /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
Body:
    `id: some_service
endpoint:
    production: someservice.com`
```

Remove a service

- Method: DELETE
- URI: /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
- URI: /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
Body:
    `id: some_service
endpoint:
    production: someservice.com`
```

Get info about a service

- Method: GET
- URI: /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
- URI: /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
- URI: /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
- URI: /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
- URI: /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
- URI: /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
- URI: /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 with an app

- Method: PUT
- URI: /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 with an app

- Method: DELETE
- URI: /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
- URI: /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 binded 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
- URI: /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
- URI: /services/instances/<serviceinstancename>/status

Returns 200 in case of success.

Example:

```
GET /services/instances/mymysql/status HTTP/1.1
```

1.4 Quotas

Get quota info of an user

- Method: GET
- URI: /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
- URI: /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
- URI: /healers/<healer>

Returns 200 in case of success.

Example:

```
GET /healers/app-heal HTTP/1.1
```

1.6 Platforms

List platforms

- Method: GET
- URI: /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 an user

- Method: POST
- URI: /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
- URI: /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
- URI: /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
- URI: /users/tokens

Returns 200 in case of success.

Example:

```
DELETE /users/tokens HTTP/1.1
```

Change password

- Method: PUT
- URI: /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 an user

- Method: DELETE
- URI: /users

Returns 200 in case of success.

Example:

```
DELETE /users HTTP/1.1
```

Add public key to user

- Method: POST
- URI: /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
- URI: /users/keys
- Body: *{"key":"my-key"}*

Returns 200 in case of success.

Example:

```
DELETE /users/keys HTTP/1.1
Body: `{"key":"my-key"}`
```

1.8 Teams

List teams

- Method: GET
- URI: /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
- URI: /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
- URI: /teams

Returns 200 in case of success.

Example:

```
POST /teams HTTP/1.1
{"name": "teamname"}
```

Remove a team

- Method: DELETE
- URI: /teams/<teamname>

Returns 200 in case of success.

Example:

```
DELELE /teams/myteam HTTP/1.1
```

Add user to team

- Method: PUT
- URI: /teams/<teanmaname>/<username>

Returns 200 in case of success.

Example:

```
PUT /teams/myteam/myuser HTTP/1.1
```

Remove user from team

- Method: DELETE
- URI: /teams/<teanmaname>/<username>

Returns 200 in case of success.

Example:

```
DELETE /teams/myteam/myuser HTTP/1.1
```

1.9 Tokens

Generate app token

- Method: POST
- URI: /tokens
- Format: json

Returns 200 in case of success, with the token in the body.

Example:

```
POST /tokens HTTP/1.1
    {
            "Token": "sometoken",
            "Creation": "2001/01/01",
            "Expires": 1000,
            "AppName": "appname",
    }
```

1.10 Deploy

Deploy list

- Method: GET
- URI: /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":29
```

Get info about a deploy

- Method: GET
- Format: json
- URI: /deploys/:deployid

Returns 200 in case of success. Returns 404 if deploy is not found.

Example:

```
GET /deploys/12345
{"App":"myapp","Commit":"e82nn93nd93mm12o2ueh83dhbd3iu112","Diff":"test_diff","Duration":10000000000,
```

7.3 Services

You can manage your services using the tsuru command-line interface.

To list all services avaliable you can use, you can use the service-list command:

```
$ tsuru service-list
```

To add a new instance of a service, use the service-add command:

```
$ tsuru service-add <service_name> <service_instance_name>
```

To remove an instance of a service, use the service-remove command:

```
$ tsuru service-remove <service_instance_name>
```

To bind a service instance with an app you can use the bind command. If this service has any variable to be used by your app, tsuru will inject this variables in the app's environment.

```
$ tsuru bind <service_instance_name> [--app appname]
```

And to unbind, use unbind command:

```
$ tsuru unbind <service_instance_name> [--app appname]
```

For more details on the `--app` flag, see "Guessing app names" section of tsuru command documentation.

7.4 tsuru-admin usage

tsuru-admin command supports administrative operations on a tsuru server. Please make sure you have it *installed* before continuing on this guide.

In order to use `tsuru-admin` commands, a user should be an *admin user*. To be an admin user you should be member of an *admin team*.

7.4.1 Setting a target

The target for the tsuru-admin command should point to the *listen* address configured in your tsuru.conf file.

```
listen: ":8080"
```

```
$ tsuru-admin target-add default tsuru.myhost.com:8080
$ tsuru-admin target-set default
```

7.4.2 Commands

All the "container*"" commands below only exist when using the docker provisioner.

container-move

```
$ tsuru-admin container-move <container id> <to host>
```

This command allow you to specify a container id and a destination host, this will create a new container on the destination host and remove the container from its previous host.

containers-move

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

It allows you to move all containers from one host to another. This is useful when doing maintenance on hosts. <from host> and <to host> must be host names of existing docker nodes.

This command will go through the following steps:

- Enumerate all units at the origin host;
- For each unit, create a new unit at the destination host;
- Erase each unit from the origin host.

containers-rebalance

```
$ tsuru-admin containers-rebalance [--dry]
```

Instead of specifying hosts as in the containers-move command, this command will automatically choose to which host each unit should be moved, trying to distribute the units as evenly as possible.

The –dry flag runs the balancing algorithm without doing any real modification. It will only print which units would be moved and where they would be created.

All the “platform*”” commands below only exist when using the docker provisioner.

docker-node-add

```
$ tsuru-admin docker-node-add [param_name=param_value]... [--register]
```

This command add a node to your docker cluster. By default, this command will call the configured IaaS to create a new machine. Every param will be sent to the IaaS implementation.

You should configure in **tsuru.conf** the protocol and port for IaaS be able to access your node (you can see it here).

If you want to just register an docker node, you should use the –register flag with an **address=http://your-docker-node:docker-port**

The command always check if your node address is accessible.

docker-node-list

```
$ tsuru-admin docker-node-list [-f/--filter <metadata>=<value>]
```

This command list all nodes present in the cluster. It will also show you metadata associated to each node and the IaaS ID if the node was added using tsuru builtin IaaS providers.

Using the `-f/--filter` flag, the user is able to filter the nodes that appear in the list based on the key pairs displayed in the metadata column. Users can also combine filters with multiple listings of `-f`:

```
$ tsuru-admin docker-node-list -f pool=mypool -f LastSuccess=2014-10-20T15:28:28-02:00
```

docker-node-remove

```
$ tsuru-admin docker-node-remove <address> [--destroy]
```

This command removes a node from the cluster. Optionally it also destroys the created IaaS machine if the `--destroy` flag is passed.

platform-add

```
$ tsuru-admin platform-add <name> [--dockerfile]
```

This command allow you to add a new platform to your tsuru installation. It will automatically create and build a whole new platform on tsuru server and will allow your users to create apps based on that platform.

The –dockerfile flag is an URL to a dockerfile which will create your platform.

platform-update

```
$ tsuru-admin platform-update <name> [-d/--dockerfile]
```

This command allow you to update a platform in your tsuru installation. It will automatically rebuild your platform and will flag apps to update platform on next deploy.

The –dockerfile flag is an URL to a dockerfile which will update your platform.

machines-list

```
$ tsuru-admin machines-list
```

This command will list all machines created using `docker-node-add` and a IaaS provider.

machine-destroy

```
$ tsuru-admin machines-list <machine id>
```

This command will destroy a IaaS machine based on its ID.

ssh

```
$ tsuru-admin ssh <container-id>
```

This command opens a SSH connection to the container, using the API server as a proxy. The user may specify part of the ID of the container. For example:

```
$ tsuru app-info -a myapp
Application: tsuru-dashboard
Repository: git@54.94.9.232:tsuru-dashboard.git
Platform: python
Teams: admin
```

```
Address: tsuru-dashboard.54.94.9.232.xip.io
Owner: admin@example.com
Deploys: 1
Units:
+-----------------------------------------------------------------+---------+
| Unit                                                            | State   |
+-----------------------------------------------------------------+---------+
| 39f82550514af3bbbec1fd204eba000546217a2fe6049e80eb28899db0419b2f | started |
+-----------------------------------------------------------------+---------+
$ tsuru-admin ssh 39f8
Welcome to Ubuntu 14.04 LTS (GNU/Linux 3.13.0-24-generic x86_64)
ubuntu@ip-10-253-6-84:~$
```

docker-healing-list

```
$ tsuru-admin docker-healing-list [--node] [--container]
```

This command will list all healing processes started for nodes or containers.

plan-create

```
$ tsuru-admin plan-create <name> -c/--cpu-share cpushare [-m/--memory memory] [-s/--swap swap] [-d/--
```

This command creates a new plan for being used when creating new apps.

The `--cpushare` flag defines a relative amount of cpu share for units created in apps using this plan. This value is unitless and relative, so specifying the same value for all plans means all units will equally share processing power.

The `--memory` flag defines how much physical memory a unit is able to use, in bytes.

The `--swap` flag defines how much virtual swap memory a unit is able to use, in bytes.

The `--default` flag sets this plan as the default plan. It means this plan will be used when creating an app without explicitly setting a plan.

plan-remove

```
$ tsuru-admin plan-remove <name>
```

This command removes an existing plan, it will no longer be available for newly created apps. However, this won't change anything for existing apps that were created using the removed plan. They will keep using the same value amount of resources described by the plan.

7.5 Client usage

After *installing the tsuru client*, you must set the target with the tsuru server URL, something like:

7.5.1 Setting a target

```
$ tsuru target-add default https://cloud.tsuru.io
$ tsuru target-set default
```

7.5.2 Authentication

After that, all you need is to create a user and authenticate to start creating apps and pushing code to them. Use create-user and login:

```
$ tsuru user-create youremail@gmail.com
$ tsuru login youremail@gmail.com
```

7.5.3 Apps

Associating your user to a team

You need to be member of a team to create an app. To create a new team, use create-team:

```
$ tsuru team-create teamname
```

Creating an app

To create an app, use app-create:

```
$ tsuru app-create myblog <platform> [--plan/-p plan_name] [--team/-t team_owner]
```

The `platform` parameter is the name of the platform to be used when creating the app. This will definer how tsuru understands and executes your app. The list of available platforms can be found running `tsuru platform-list`.

The `--plan` parameter defines the plan to be used. The plan specifies how computational resources are allocated to your application. Typically this means limits for memory and swap usage, and how much cpu share is allocated. The list of available plans can be found running `tsuru plan-list`.

If this parameter is not informed, tsuru will choose the plan with the `default` flag set to true.

The `team` parameter describes which team is responsible for the created app, this is only needed if the current user belongs to more than one team, in which case this parameter will be mandatory.

After running successfully the command will return your app's remote url, you should add it to your git repository:

```
$ git remote add tsuru git@tsuru.myhost.com:myblog.git
```

Listing your apps

When your app is ready, you can push to it. To check whether it is ready or not, you can use app-list:

```
$ tsuru app-list
```

This will return something like:

```
+-------------+-------------------------+-------------------------------------------+
| Application | Units State Summary     | Ip                                        |
+-------------+-------------------------+-------------------------------------------+
| myblog      | 1 of 1 units in-service | myblog-838381.us-east-1-elb.amazonaws.com |
+-------------+-------------------------+-------------------------------------------+
```

Showing app info

You can also use the app-info command to view information of an app. Including the status of the app:

```
$ tsuru app-info
```

This will return something like:

```
Application: myblog
Platform: gunicorn
Repository: git@githost.com:myblog.git
Teams: team1, team2
Units:
+----------+---------+
| Unit     | State   |
+----------+---------+
| myblog/0 | started |
| myblog/1 | started |
+----------+---------+
```

tsuru uses information from git configuration to guess the name of the app, for more details, see "Guessing app names" section of tsuru command documentation.

7.5.4 Public Keys

You can try to push now, but you'll get a permission error, because you haven't pushed your key yet.

```
$ tsuru key-add
```

This will search for a *id_rsa.pub* file in ~/.ssh/, if you don't have a generated key yet, you should generate one before running this command.

If you have a public key in other format (for example, DSA), you can also give the public key file to key-add:

```
$ tsuru key-add $HOME/.ssh/id_dsa.pub
```

After your key is added, you can push your application to your cloud:

```
$ git push tsuru master
```

7.5.5 Running commands

After that, you can check your app's url in the browser and see your app there. You'll probably need to run migrations or other deploy related commands. To run a single command, you should use the command run:

```
$ tsuru run "python manage.py syncdb && python manage.py migrate"
```

7.5.6 Further instructions

For a complete reference, check the documentation for tsuru command: http://godoc.org/github.com/tsuru/tsuru-client/tsuru.

CHAPTER 8

Frequently Asked Questions

- How does environment variables work?
- How does the quota system works?
- How routing works?
- How are Git repositories managed?

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.

8.1 How does 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.

8.2 How does the quota system works?

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.

8.3 How routing works?

tsuru has a router interface, which makes extremely easy to change the way routing works with any provisioner. There are two ready-to-go routers: one using hipache and another with elb.

8.4 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.

8.5 Client installation fails with "undefined: bufio.Scanner". What does it mean?

tsuru clients require Go 1.1 or later. The message `undefined: bufio.Scanner` means that you're using an old version of Go. You'll have to install the last version.

If you're using Homebrew on Mac OS, just run:

```
$ brew update
$ brew upgrade go
```

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 9

Release notes

Release notes for the official tsuru releases. Each release note will tell you what's new in each version.

9.1 tsr

tsr is the tsuru server daemon.

9.1.1 0.8.1 release

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.

9.1.2 0.8.0 release

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).

9.1.3 0.7.2 release

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;

9.1.4 0.7.1 release

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;

9.1.5 0.7.0 release

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);
})
```

9.1.6 0.6.2 release

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.

9.1.7 0.6.1 release

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.

9.1.8 0.6.0 release

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

9.1.9 0.5.3 release

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.

9.1.10 0.5.2 release

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

9.1.11 0.5.1 release

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.

9.1.12 0.5.0 release

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 bellow, 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 bellow:

```
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.

9.1.13 0.4.0 release

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.

9.1.14 0.3.12 release

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.

9.1.15 0.3.11 release

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);
    }
);
```

9.1.16 0.3.10 release

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.

9.1.17 0.3.9 release

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.

9.1.18 0.3.8 release

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.

9.1.19 0.3.7 release

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.

9.1.20 0.3.6 release

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.

9.1.21 0.3.5 release

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.

9.1.22 0.3.4 release

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.

9.1.23 0.3.3 release

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.

9.1.24 0.3.2 release

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.

9.1.25 0.3.1 release

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

9.1.26 0.3.0 release

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>
```

9.2 tsuru

tsuru is the tsuru client. For details on releases of the client, check the release history in the tsuru-client repository at GitHub.

9.3 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.

9.4 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.