tsuru Documentation

Release 0.1

timeredbull

June 25, 2014

Contents

CHAPTER 1

What is Tsuru?

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.

Learn more in *Tsuru's Overview* or check out our *FAQ*.

- *Build your own PaaS with Tsuru*
- *Deploy your application on Tsuru*
- *Provide services on Tsuru*

CHAPTER 2

More documentation

2.1 For tsuru users

- *clients installation guide*
- *tsuru client usage guide*
- *using services*
- *building your application*

2.2 For tsuru ops

- *build your own PaaS using juju*
- *build your own PaaS using docker*
- *tsuru configuration*
- *backing up tsuru*
- *tsuru api reference*
- *building your service tutorial*
- *crane usage guide*
- *tsuru services api workflow*

2.3 Contributions and Feedback

- *how to contribute*
- *coding style*
- *setting up your tsuru development environment*
- *community*

2.3.1 API reference

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-Legth: 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-Legth: 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 hte response containing the statusn 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 enviroment variables

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

Returns 200 in case of success, and json in the body returning a dictionary with enviroment names and values..

Example:

```
GET /apps/myapp/env HTTP/1.1
{"DATABASE_HOST":"localhost"}
```

Set an app enviroment

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

Returns 200 in case of success.

Example:

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

Delete an app enviroment

- 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
```

1.2 Services

List services

- Method: GET
- URI: /services
- Format: json

Returns 200 in case of success.

Example:

```
GET /services HTTP/1.1
Content-Legth: 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

1.x 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-Legth: 29
{"items": 10, "available": 2}
```

1.x 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-Legth: 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.x 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-Legth: 67
[{Name: "python"},{Name: "java"},{Name: "ruby20"},{Name: "static"}]
```

1.x 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
```

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.x 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-Legth: 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
```

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.x 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",
    }
```

2.3.2 Build your own PaaS

This document describes how to create a private PaaS service using tsuru. It contains instructions on how to build tsuru and some of its components from source.

This document assumes that tsuru is being installed on a Ubuntu machine. You can use equivalent packages for beanstalkd, git, MongoDB and other tsuru dependencies. Please make sure you satisfy minimal version requirements.

There's also a contributed Vagrant box, that setups a PaaS using Chef. You can check this out: https://github.com/hfeeki/vagrant-tsuru.

Overview

The Tsuru PaaS is composed by multiple components:

- tsuru server
- tsuru collector
- gandalf
- charms

And these components have their own dependencies, like:

- mongodb (>=2.2.0)
- beanstalkd (>=1.4.6)
- git-daemon (git>=1.7)
- juju (python version, >=0.5)
- libyaml (>=0.1.4)

Requirements

1. Operating System

At the moment, tsuru server is fully supported and tested on Ubuntu 12.04 and the steps below will guide you throught the install process.

If you try to build tsuru server on most Linux systems, you should have few problems and if there are problems, we are able to help you. Just ask on #tsuru channel on irc.freenode.net.

- *Have you tried tsuru server on other systems? Let us know and contribute to the project.*

2. Hardware

Tsuru server is a lightweight framework and can be run in a single small machine along with all the deps.

3. Software

3.1 MongoDB

Tsuru needs MongoDB stable, distributed by 10gen. It's pretty easy to get it running on Ubuntu

3.2 Juju

Tsuru uses juju to orchestrates your "apps". To install juju follow the juju install guide. Please make sure that you configure Juju properly. Then run:

```
$ juju bootstrap
```

Juju Charms define how platforms will be installed. You may take a look at juju charms collection or use the charms provided by tsuru

Put it somewhere and define the setting `juju:charms-path` in the configuration file:

```
$ git clone git://github.com/globocom/charms.git /home/me/charms
$ cat /etc/tsuru/tsuru.conf
# ...
juju:
  charms-path: /home/me/charms
```

3.3 Beanstalkd

Tsuru uses Beanstalkd as a work queue. Install the latest version, by doing this:

```
$ sudo apt-get install -y beanstalkd
```

3.4 Gandalf

Tsuru uses Gandalf to manage git repositories, to get it installed follow this steps

Installing pre-built binaries

You can download pre-built binaries of tsuru and collector. There are binaries available only for Linux 64 bits, so make sure that `uname -m` prints `x86_64`:

```
$ uname -m
x86_64
```

Then download and install the tsr binary:

```
$ curl -sL https://s3.amazonaws.com/tsuru/dist-server/tsr.tar.gz | sudo tar -xz -C /usr/bin
```

These commands will install `tsr` in `/usr/bin` (you will need to be a sudoer and provide your password). You may install this command in your `PATH`.

Installing from source

0. Build dependencies

To build tsuru from source you will need to install the following packages

```
$ sudo apt-get install -y golang-go git mercurial bzr gcc
```

1. Install the tsuru tsr

Add the following lines to your ~/.bashrc:

```
$ export GOPATH=/home/ubuntu/.go
$ export PATH=${GOPATH}/bin:${PATH}
```

Then execute:

```
$ source ~/.bashrc
$ go get github.com/globocom/tsuru/tsr
```

Configuring tsuru

Before running tsuru, you must configure it. By default, tsuru will look for the configuration file in the `/etc/tsuru/tsuru.conf` path. You can check a sample configuration file and documentation for each tsuru setting in the *"Configuring tsuru"* page.

You can download the sample configuration file from Github:

```
$ [sudo] mkdir /etc/tsuru
$ [sudo] curl -sL https://raw.github.com/globocom/tsuru/master/etc/tsuru.conf -o /etc/tsuru/tsuru.co
```

Make sure you define the required settings (database connection, authentication configuration, AWS credentials, etc.) before running tsuru.

Running tsuru

Now that you have `tsr` properly installed, and you *configured tsuru*, you're three steps away from running it.

1. Start mongodb

```
$ sudo service mongodb  start
```

2. Start beanstalkd

```
$ sudo service beanstalkd start
```

3. Start api and collector

```
$ tsr api &
$ tsr collector &
```

One can see the logs in:

```
$ tail -f /var/log/syslog
```

Using tsuru

Congratulations! At this point you should have a working tsuru server running on your machine, follow the *tsuru client usage guide* to start build your apps.

2.3.3 community

irc channel

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

ticket system

ticket system - report bugs and make feature requests

2.3.4 Configuring tsuru

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

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>
```

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. Omiting these settings won't break Tsuru, but users would not be able to reset their password automattically.

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 its own authentication mechanism, that hashes passwords brcypt. Tokens are generated during authentication, and are hashed using SHA512.

This mechanism requires two settings to operate: `auth:hash-cost` and `auth:token-expire-days`. Each setting is described below.

The `auth` section also controls whether user registation is on or off. When user registration is off, the user creation URL is not registered in the server.

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

auth:hash-cost 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 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".

Amazon Web Services (AWS) configuration

Tsuru is able to use Amazon Web Services (AWS) Simple Storage Service (S3) to provide static storage for apps. Whenever `bucket-support` is true, Tsuru will create a S3 bucket and AWS Identity and Access Management (IAM) credentials to access this bucket during the app creation process. In order to be able to comunicate with AWS API's, tsuru needs some settings, listed below.

For more details on AWS authentication, AWS AIM and AWS S3, check AWS docs: https://aws.amazon.com/documentation/.

bucket-support `bucket-support` is a boolean flag, that turns on the bucket per app feature. This field is optional, and defaults to false.

aws:access-key-id `aws:access-key-id` is the access key ID used by tsuru to authenticate with AWS API. Given that `bucket-support` is true, this setting is required and has no default value.

aws:secret-access-key `aws:secret-access-key` is the secret access key used by tsuru to authenticate with AWS API. Given that `bucket-support` is true, this setting is required and has no default value.

aws:ec2:endpoint `aws:ec2:endpoint` is the EC2 endpoint that tsuru will call to communicate with ec2. It's only used for *juju* healers.

aws:iam:endpoint `aws:iam:endpoint` is the IAM endpoint that tsuru will call to create credentials for its applications. This setting is optional, and defaults to `https://iam.amazonaws.com/`. You should change this setting only when using another service that also implements IAM's API.

aws:s3:region-name `aws:s3:region-name` is the name of the region that tsuru will use to create S3 buckets. Given that `bucket-support` is true, this setting is required and has no default value.

aws:s3:endpoint `aws:s3:endpoint` is the S3 endpoint that tsuru will call to create buckets for its applications. Given that `bucket-support` is true, this setting is required and has no default value.

aws:s3:location-constraint `aws:s3:location-constraint` indicates whether buckets should be stored in the selected region. Given that `bucket-support` is true, this setting is required and has no default value.

For more details, check the documentation for buckets and regions: http://docs.aws.amazon.com/AmazonS3/latest/dev/LocationSelection

aws:s3:lowercase-bucket `aws:s3:lowercase-bucket` will be true if the region requires bucket names to be lowercase. Given that `bucket-support` is true, this setting is required and has no default value.

queue configuration

Tsuru uses a work queue for asynchronous tasks. By default it will use beanstalkd. You can customize the used queue, and settings related to the queue (like the address where beanstalkd is listening).

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 is optional and defaults to "beanstalkd".

queue-server `queue-server` is the TCP address where beanstalkd is listening. This setting is optional and defaults to "localhost:11300".

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".

Defining the provisioner

Tsuru supports multiple provisioners. A provisioner is a Go type that satisfies an interface. By default, tsuru will use `JujuProvisioner` (identified by the string "juju"). To use other provisioner, that has been already registered with tsuru, one must define the setting `provisioner`.

provisioner `provisioner` is the string the name of the provisioner that will be used by tsuru. This setting is optional and defaults to "juju".

You can also configure the provisioner (check the next section for details on Juju configuration).

Juju provisioner configuration

"juju" is the default provisioner used by Tsuru. It's named after the tool used by tsuru to provision and manage instances. It's a extended version of Juju, supporting Amazon's Virtual Private Cloud (VPC) and Elastic Load Balancing (ELB).

Charms path

Juju describe services as Charms. Each tsuru platform is a Juju charm. The tsuru team provides a collection of charms with customized hooks: https://github.com/globocom/charms. In order (for more details, refer to *build documentation*).

juju:charms-path `charms-path` is the path where tsuru should look for charms when creating new apps. If you specify the value "/etc/juju/charms", your charms tree should look something like this:

```
.
-- centos
|   -- ...
-- precise
    -- go
    |   -- config.yaml
    |   -- hooks
    |   ...
    |   -- metadata.yaml
    -- nodejs
    |   -- config.yaml
```

```
|   -- hooks
|   ...
|   -- metadata.yaml
-- python
|   -- config.yaml
|   -- hooks
|   ...
|   -- metadata.yaml
|   -- utils
|       -- circus.ini
|       -- nginx.conf
-- rack
|   -- config.yaml
|   -- hooks
|   ...
|   -- metadata.yaml
-- ruby
|   -- config.yaml
|   -- hooks
|   ...
|   -- metadata.yaml
-- static
    -- config.yaml
    -- hooks
    ...
    -- metadata.yaml
```

Given that you're using juju, this setting is mandatory and has no default value.

Storing units in the database

Juju provisioner uses the database to store information about units. It uses a MongoDB collection that will be located in the same database used by tsuru. One can set the name of this collection using the setting described below:

juju:units-collection `juju:units-collection` defines the name of the collection that Juju provisioner should use to store information about units. This setting is required by the provisioner and has no default value.

Elastic Load Balancing support

Juju provisioner can manage load balancers per app using Elastic Load Balancing (ELB) API, provided by Amazon. In order to enable Elastic Load Balancing support, one must set `juju:use-elb` to true and define other settings described below:

juju:use-elb `juju:use-elb` is a boolean flag that indicates whether Juju provisioner will use ELB. When enabled, it will create a load balancer per app, registering and deregistering units as they come and go, and deleting the load balancer when the app is removed. This setting is optional and defaults to false.

Whenever `juju:use-elb` is defined to be true, other settings related to load balancing become mandatory: `juju:elb-endpoint`, `juju:elb-collection`, `juju:elb-avail-zones` (or `juju:elb-vpc-subnets` and `juju:elb-vpc-secgroups`, see `juju:elb-use-vpc` for more details).

juju:elb-endpoint `juju:elb-endpoint` is the ELB endpoint that tsuru will use to manage load balancers. This setting has no default value, and is mandatory once `juju:use-elb` is true. When `juju:use-elb` is false, the value of this setting is irrelevant.

juju:elb-collection `juju:elb-collection` is the name of the collection that Juju provisioner will use to store information about load balancers.

This setting has no default value, and is mandatory once `juju:use-elb` is true. When `juju:use-elb` is false, the value of this setting is irrelevant.

juju:elb-use-vpc `juju:elb-use-vpc` is another boolean flag. It indicates whether load balancers should be created using an Amazon Virtual Private Cloud. When this setting is true, one must also define `juju:elb-vpc-subnets` and `juju:elb-vpc-secgroups`.

This setting is optional, defaults to false and has no effect when `juju:use-elb` is false.

juju:elb-vpc-subnets `juju:elb-vpc-subnets` contains a list of subnets that will be attached to the load balancer. This setting must be defined whenever `juju:elb-use-vpc` is true. It has no default value.

juju:elb-vpc-secgroups `juju:elb-vpc-secgroups` contains a list of security groups from which the load balancer will inherit rules. This setting must be defined whenever `juju:elb-use-vpc` is true. It has no default value.

juju:elb-avail-zones `juju:elb-avail-zones` contains a list of availability zones that the load balancer will communicate with. This setting has no effect when `juju:elb-use-vpc` is true, has no default value and must be defined whenever `juju:elb-use-vpc` is false.

Sample file

Here is a complete example, with S3, VPC, HTTP/TLS and load balacing enabled:

```
listen: ":8080"
use-tls: true
tls:
  cert-file: /etc/tsuru/tls/cert.pem
  key-file: /etc/tsuru/tls/key.pem
host: http://10.19.2.238:8080
database:
  url: 127.0.0.1:27017
  name: tsuru
git:
  unit-repo: /home/application/current
  host: gandalf.tsuru.io
  port: 8000
  protocol: http
auth:
  token-expire-days: 14
bucket-support: true
aws:
  access-key-id: access-key
  secret-access-key: s3cr3t
  iam:
    endpoint: https://iam.amazonaws.com/
```

```
  s3:
    region-name: sa-east-1
    endpoint: https://s3.amazonaws.com
    location-constraint: true
    lowercase-bucket: true
provisioner: juju
queue-server: "127.0.0.1:11300"
admin-team: admin
juju:
  charms-path: /etc/juju/charms
  units-collection: j_units
  use-elb: true
  elb-endpoint: https://elasticloadbalancing.amazonaws.com
  elb-collection: j_lbs
  elb-use-vpc: true
  elb-vpc-subnets:
    - subnet-a1a1a1
  elb-vpc-secgroups:
    - sg-a1a1a1
```

2.3.5 contribute

- Source hosted at GitHub
- Report issues on GitHub Issues

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

development environment

See this guide to *setting up you tsuru development environment*.

And follow our *coding style guide*.

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
```

writing docs

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

building docs

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

```
$ make doc
```

2.3.6 Build your own PaaS with tsuru and Docker

This document describes how to create a private PaaS service using tsuru and docker.

This document assumes that tsuru is being installed on a Ubuntu (12.04+) machine. You can use equivalent packages for beanstalkd, git, MongoDB and other tsuru dependencies. Please make sure you satisfy minimal version requirements.

You can use the scripts bellow to quick install tsuru with docker:

```
$ curl -O https://raw.github.com/globocom/tsuru/master/misc/docker-setup.bash; bash docker-setup.bash
```

Or follow this steps:

docker

Install docker:

```
$ sudo sh -c "echo 'deb http://ppa.launchpad.net/dotcloud/lxc-docker/ubuntu precise main' >> /etc/apt
$ sudo apt-get update -y
$ sudo apt-get install lxc-docker -y
```

MongoDB

Tsuru needs MongoDB stable, distributed by 10gen. It's pretty easy to get it running on Ubuntu

Beanstalkd

Tsuru uses Beanstalkd as a work queue. Install the latest version, by doing this:

```
$ sudo apt-get install -y beanstalkd
```

Configuring beanstalkd to start, edit the */etc/default/beanstalkd* and uncomment this line:

```
START=yes
```

Then start beanstalkd:

```
$ sudo service beanstalkd start
```

Gandalf

Tsuru uses Gandalf to manage git repositories, to get it installed follow this steps

Installing git

```
$ sudo apt-get install git -y
```

Creating git user

```
$ sudo useradd git
```

Creating directories for repositories and template

Let's create the directory for bare repositories:

```
$ sudo mkdir -p /var/repositories
$ sudo chown -R git:git /var/repositories
```

And the directory for template and add the tsuru hooks:

```
$ sudo mkdir -p /home/git/bare-template/hooks
$ curl https://raw.github.com/globocom/tsuru/master/misc/git-hooks/post-receive > /home/git/bare-temp
$ sudo chown -R git:git /home/git/bare-template
```

Configuring gandalf

```
sudo bash -c 'echo "bin-path: /usr/bin
database:
  url: 127.0.0.1:27017
  name: gandalf
git:
  bare:
    location: /var/repositories
    template: /home/git/bare-template
  daemon:
    export-all: true
host: localhost
webserver:
  port: \":8000\"" > /etc/gandalf.conf'
```

Change the 'host: localhost' to your base domain.

Tsuru api and collector

You can download pre-built binaries of tsuru and collector. There are binaries available only for Linux 64 bits, so make sure that `uname -m` prints `x86_64`:

```
$ uname -m
x86_64
```

Then download and install the tsr binary:

```
$ curl -sL https://s3.amazonaws.com/tsuru/dist-server/tsr.tar.gz | sudo tar -xz -C /usr/bin
```

These commands will install `tsr` command in `/usr/bin` (you will need to be a sudoer and provide your password). You may install this command in your `PATH`.

Configuring

Before running tsuru, you must configure it. By default, tsuru will look for the configuration file in the `/etc/tsuru/tsuru.conf` path. You can check a sample configuration file and documentation for each tsuru setting in the *"Configuring tsuru"* page.

You can download the sample configuration file from Github:

```
$ [sudo] mkdir /etc/tsuru
$ [sudo] curl -sL https://raw.github.com/globocom/tsuru/master/etc/tsuru-docker.conf -o /etc/tsuru/ts
```

By default, this configuration will use the tsuru image namespace, so if you try to create an application using python platform, tsuru will search for an image named tsuru/python. You can change this default behavior by changing the docker:repository-namespace config field.

Running

Now that you have `tsr` properly installed, and you *configured tsuru*, you're three steps away from running it.

Start api and collector

```
$ tsr collector &
$ sudo tsr api &
```

You can see the logs in:

```
$ tail -f /var/log/syslog
```

Using tsuru Congratulations! At this point you should have a working tsuru server running on your machine, follow the *tsuru client usage guide* to start build your apps.

2.3.7 Download

Client binaries

tsuru clients are also distributed in binary version, so you can just download an executable and put them somewhere in your `PATH`.

It's important to note that all binaries are platform dependent. Currently, we provide each of them in three flavors:

1. **darwin_amd64**: This is Mac OS X, 64 bits. Make sure the command `uname -ms` prints "Darwin x86_64", otherwise this binary will not work in your system;
2. **linux_386**: This is Linux, 32 bits. Make sure the command `uname -ms` prints "Linux x86", otherwise this binary will not work in your system;
3. **linux_amd64**: This is Linux, 64 bits. Make sure the command `uname -ms` prints "Linux x86_64", otherwise this binary will not work in your system.

Below are the links to the binaries, you can just download, extract the archive and put the binary somewhere in your PATH:

darwin_amd64

- tsuru: https://s3.amazonaws.com/tsuru/dist-cmd/tsuru-darwin-amd64.tar.gz
- tsuru-admin: https://s3.amazonaws.com/tsuru/dist-cmd/tsuru-admin-darwin-amd64.tar.gz
- crane: https://s3.amazonaws.com/tsuru/dist-cmd/crane-darwin-amd64.tar.gz

linux_386

- tsuru: https://s3.amazonaws.com/tsuru/dist-cmd/tsuru-linux-386.tar.gz
- tsuru-admin: https://s3.amazonaws.com/tsuru/dist-cmd/tsuru-admin-linux-386.tar.gz
- crane: https://s3.amazonaws.com/tsuru/dist-cmd/crane-linux-386.tar.gz

linux_amd64

- tsuru: https://s3.amazonaws.com/tsuru/dist-cmd/tsuru-linux-amd64.tar.gz
- tsuru-admin: https://s3.amazonaws.com/tsuru/dist-cmd/tsuru-admin-linux-amd64.tar.gz
- crane: https://s3.amazonaws.com/tsuru/dist-cmd/crane-linux-amd64.tar.gz

2.3.8 Tsuru Frequently Asked Questions

This document is an attempt to explain concepts you'll face when deploying and managing applications using Tsuru. To request additional explanations you can open an issue on our issue tracker, talk to us at #tsuru @ freenode.net or open a thread on our mailing list.

What is Tsuru?

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.

What is an application?

An application, in Tsuru, is a program's source code, dependencies list - on operational system and language level - and a Procfile with instructions on how to run that program. An application has a name, a unique address, a Platform, associated development teams, a repository and a set of units.

What is a unit?

A unit is an isolated Unix container or a virtual machine - depending on the configured provisioner. 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 applications processes defined on the Procfile.

What is a platform?

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, or a virtual machine 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 in Tsuru, but currently not managed by it, all Tsuru does (by now) is to keep database records for each existent platform. Every application runs on top of a platform.

What is a service?

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 its creators.

Check the *service usage documentation* for more on using services and the *building your own service tutorial* for a quick start on how to extend Tsuru by creating new services.

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.

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.

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.

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.

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 verson.

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

```
% brew update
% brew upgrade go
```

2.3.9 Tsuru Overview

This document is in alpha state, to suggest improvements check out the related github issue.

Tsuru is an open source PaaS. If you don't know what a PaaS is and what it does, see wikipedia's description.

It follows the principles described in the The Twelve-Factor App methodology.

Fast and easy deployment

Deploying an app is simple and easy. No special tools needed, just a plain git push. The entire process is very simple, especially from the second deployment, whether your app is big or small.

Tsuru uses git as the means of deploying an application. You don't need master git in order to deploy an app to Tsuru, although you will need to know the very basic workflow, add/commit/push and remote managing. Git allows really fast deploys, and Tsuru makes the best possible use of it by not cloning the whole repository history of your application, there's no need to have that information in the application webserver.

Tsuru will also take care of all the applications dependencies in the deployment process. You can specify operating system and language specific dependencies. For example, if you have a Python application, tsuru will search for the requirements.txt file, but first it will search for OS dependencies (a list of deb packages in a file named requirements.apt, in the case of Ubuntu).

Tsuru also has hooks that can trigger commands before and after some events that happen during the deployment process, like restart (represented by `pre-restart` and `post-restart` hooks).

Continuous Deployment

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

Add-on Resources

Instantly provision and integrate third party services with one command. Tsuru provides the basic services your application will need, like searching, caching, storage and frontend; you can get all of that in a fashionable and really easy way using Tsuru's command line.

Per-Environment Config Variables

Configuration for an application should be stored in environment variables - and we know that. Tsuru lets you define your environment variables using the command line, so you can have the configuration flexibility your application need.

Tsuru also makes use of environment variables. When you bind a service with your application, Tsuru gives the service the ability to inject environment variables in your application environment. For instance, if you use the default MySQL service, it will inject variables for you to establish a connection with your application database.

Custom Services

Tsuru already has services for you to use, but you don't need to use them at all if you don't want to. If you already have, let's say, a MySQL server running on your infrastructure, all you need to do in order to use it is simply configure environment variables and use them in your application config.

You can also create your own services and make them available for you and others to use it on Tsuru. It's so easy to do so that you'll want to sell your own services. Tsuru talks with services using a well defined API, all you have to do is implement four endpoints that knows how to provision instances of your services and bind them to tsuru applications (like creating VMs, authorizing security groups, creating ACLs, etc), and register your service in Tsuru with a really simple yaml manifest.

Logging and Visibility

Full visibility into your app's operations with real-time logging, process status inspection, and an audit trail of all releases. Tsuru will capture standard streams (output and error) from your application and expose them via the `tsuru log` command. You can also filter logs, for example, if you don't want to see the logs of developers activity (e.g.: a deploy action), you can specify the source as "app" and you'll get only the application webserver logs.

Process Management

Tsuru manages all processes from an application, so you don't have to worry about it. But it does not know to start it. You'll have to teach Tsuru how to start your application using a Procfile. Tsuru reads the Procfile and uses Circus to start and manage the running process. You can even enable a web console for Circus to manage your application process and to watch CPU and memory usage in real-time through a web interface.

Tsuru also allows you to easily restart your application process via command line. Although Tsuru will do all the hard work of managing and fixing eventual problems with your process, you might need to restart your application manually, so we give you an easy way to do it.

Control Surfaces

Tsuru exposes its features through a solid, stable REST API. You can write clients for this API, or you can use one of the clients maintained by tsuru developers.

Tsuru ships with two API clients: the command line interface (CLI), which is pretty stable and ready for day-to-day usage; and the web interface, which is under development, but is also a great tool to manage, check logs and monitor applications and services resources.

Scaling

The Juju provisioner allows you to easily add and remove units, enabling one to scale an application painlessly. It will take care of the application code replication, and services binding. There's nothing required to the developer to do in order to scale an application, just add a new unit and Tsuru will do the trick.

You may also want to scale using the Front end as a Service, powered by Varnish. One single application might have a whole farm of Varnish VMs in front of it handling all the traffic.

Built-in Database Services

Tsuru already has a variety of database services available for setup on your cloud. It allows you to easily create a service instance for your application usage and bind them together. The service setup for your application is transparent

by the use of environment variables, which are exported in all instances of the application, allowing your configuration to fit several environments (like development, staging, production, etc.)

Extensible Service and Platform Support

Tsuru allows you to easily add support for new services and new platforms. For application platforms, it uses Juju Charms, for services, it has an *API* that it uses to comunicate with them.

Collaboration

Manage sharing and deployment of your application. Tsuru uses teams to control access to resources. A developer may create a team, grant/revoke app access to/from a team or add/remove new users to/from a team. One can be a member of multiple teams and control which applications each team has access to.

Easy Server Deployment

Tsuru itself is really easy to deploy and manage, you can get it done by following these simple steps.

Distributed and Extensible

Tsuru server is easily extensible, distributed and customizable. It has the concept of `Provisioner`: a provisioner is a component that takes care of the orchestration (VM/container management) and provisioning. By default, it will deploy applications using the Juju provisioner, but you can easily implement your own provisioner and use whatever backend you wish.

When you extend Tsuru, you are able to pratically build a new PaaS in terms of behavior of provision and orchestration, making use of the great Tsuru structure. You change the whole Tsuru workflow by implementing a new provisioner.

Dev/Ops Perspective

Tsuru's components are distributed, it is composed by many pieces of software, each one made to be easily deployable and maintenable. #TODO link architecture overview.

Application Developer Perspective

We aim to make developers life easier. #TODO link development workflow.

2.3.10 Why Tsuru?

This document aims to show Tsuru's most killing features. Additionally, provides a comparison of Tsuru with others PaaS's on the market.

Easy Server Installation

It's really easy to have a running PaaS with Tsuru. We provide a serie of scripts, each one built to install and configure the required components for each Tsuru provisioner, you can check our scripts on Tsuru repository, there are separeted scripts to install each component, so it's easy to create your own script to configure a new provisioner or to change the configuration of an existing one.

But it's okay if you want more control and do not want to use our scripts, or want to better understand the interaction between Tsuru components, we built a guide only for you.

Platforms Extensibility

One of Tsuru main goals is to be easily extensible. The platform is one great example of accomplishment on that. Tsuru platforms works slightly different for each provisioner. Juju use *charms* for platform provisioning you can find the scripts on our charms repository. The Docker provisioner is a bit different, it has an specific image for each platform, if one wants to create a new platform, just extend tsuru/base image and follow the directory tree structure, the scripts and Dockerfile for our existing platforms images can be found on our images repository

Services Creation and Extension

Most applications need a service to work properly, like a database service. Tsuru provides an interface API to communicate with services APIs, but it doesn't manage services directly, this provides more control over the service and its management.

In order to create a new service you simply write an API implementing the predefined endpoints. Tsuru will call when a user performs an action using the client, read more on the building your service tutorial.

You can either create a new service or modify an existing one, if its source is open. All services APIs made by Tsuru team are open and contributions are very welcome. For example, the mongoDB api shares one database installation with everyone that is using it, if you don't like it and want to change it, you can do it and create a new service on Tsuru with your own implementation.

IaaS's and Provisioners

Tsuru provides an easy way to change the application unit provisioning system and it already has two working provisioners, Juju, Docker. But the main advantage is the ease of extending the provisioning system. One can simply implement the Provision interface Tsuru provides, configure it on your installation and start using it.

Routers

Tsuru also provides an abstraction for routing control and load balancing in application units. It provides a routing interface, that you can combine on many ways: you can plug any router with any provisioner, you can also create your own routing backend and plug it with any existing provisioner, this can be done only changing Tsuru's configuration file.

Comparing Tsuru With Other PaaS's

The following table compares Tsuru with OpenShift and Stackato PaaS's.

If you have anything to consider, or want to ask us to add another PaaS on the list contact us in #tsuru @ freenode.net or at our mailing list

	Tsuru	OpenShift	Stackato
Built-in Platforms	Node.js, PHP, HTML, Python, Ruby, Go, Java	Java, PHP, Ruby, Node.js, Python	Java, Node.Js, Perl, PHP, Python, Ruby
End-user web UI	Yes (Abyss)	Yes	Yes
CLI	Yes	Yes	Yes
Deployment hooks	Yes	No	Yes
SSH Access	Yes (management-only)	Yes	Yes
Run Commands Remotely	Yes	No	No
Application Monitoring	Yes	Yes	Yes
SQL Databases	MySQL	MySQL, PostgreSQL	MySQL, PostgreSQL
NoSQL Databases	MongoDB, Cassandra Memcached, Redis	MongoDB	MongoDB, Redis
Log Streaming	Yes	Yes (not built-in)	Yes
Metering/Billing API	No (issue 466)	No	Yes
Quota System	Yes	Yes	Yes
Container Based Apps	Yes	Yes	Yes
VMs Based Apps	Yes	No	No
Open Source	Yes	Yes	No
Free	Yes	Yes	Yes
Paid/Closed Version	No	Yes	Yes
PaaS Healing	Yes	No	No
App Healing	Yes	No	No
App Fault Tolerance	Yes	Yes (by cartridge)	Yes
Auto Scaling	No (issue 154)	Yes	Yes (for some IaaS's)
Manual Scaling	Yes	No	Yes

2.3.11 Application Deployment

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

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>.

2.3.12 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 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.

Install the tsuru client

Install the Tsuru client for your development platform.

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

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.

Login

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

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

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*

2.3.13 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>.

2.3.14 Setting up you tsuru development environment

To install tsuru from source, you need to have Go installed and configured. This file will guide you through the necessary steps to get tsuru's development environment.

Installing Go

You need to install the last version of Go to compile tsuru. You can download binaries distribution from Go website or use your preferred package installer (like Homebrew on Mac OS and apt-get on Ubuntu):

```
$ [sudo] apt-get install golang
```

```
$ brew install go
```

Installing MongoDB

tsuru uses MongoDB (+2.2), so you need to install it. For that, you can follow instructions on MongoDB website and download binary distributions (http://www.mongodb.org/downloads). You can also use your preferred package installer:

```
$ sudo apt-key adv --keyserver keyserver.ubuntu.com --recv 7F0CEB10
$ sudo bash -c 'echo "deb http://downloads-distro.mongodb.org/repo/ubuntu-upstart dist 10gen" > /etc/
$ sudo apt-get update
$ sudo apt-get install mongodb-10gen -y
```

```
$ brew install mongodb
```

Installing Beanstalkd

Tsuru uses Beanstalkd as a work queue. Install the latest version, by doing this:

```
$ sudo apt-get install -y beanstalkd
```

```
$ brew install beanstalkd
```

Installing Redis

One of Tsuru routing providers uses Redis to store information about frontends and backends. You will also need to install it:

```
$ sudo apt-get install -y redis-server
```

```
$ brew install redis
```

Installing git, bzr and mercurial

tsuru depends on go libs that use git, bazaar and mercurial, so you need to install these two version control systems to get and compile tsuru from source.

To install git, you can use your package installer:

```
$ sudo apt-get install git
```

```
$ brew install git
```

To install bazaar, follow the instructions in bazaar's website (http://wiki.bazaar.canonical.com/Download), or use your package installer:

```
$ sudo apt-get install bzr
```

```
$ brew install bzr
```

To install mercurial, you can also follow instructions on its website (http://mercurial.selenic.com/downloads/) or use your package installer:

```
$ sudo apt-get install mercurial
```

```
$ brew install mercurial
```

Setting up GOPATH and cloning the project

Go uses an environment variable called GOPATH to allow users to develop using the go build tool (http://golang.org/cmd/go). So you need to setup this variable before cloning and installing tsuru. You can set this variable to your $HOME directory, or something like *$HOME/gocode*.

Once you have defined the GOPATH variable, then run the following commands:

```
$ mkdir -p $GOPATH/src/github.com/globocom
$ cd $GOPATH/src/github.com/globocom
$ git clone git://github.com/globocom/tsuru
```

If you have already cloned the repository, just move the cloned directory to *$GOPATH/src/github.com/globocom*.

For more details on GOPATH, please check this url: http://golang.org/cmd/go/#GOPATH_environment_variable

Starting Redis, Beanstalkd and MongoDB

Before building the code and running the tests, execute the following commands to start Redis, Beanstalkd and MongoDB processes.

```
$ redis-server
$ mongod
$ beanstalkd -l 127.0.0.1
```

Installing tsuru dependencies and running 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
```

2.3.15 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 or building them from source.

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 globocom/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 1.1 or higher. Make sure you have the last version of Go installed in your system.

Pre-built binaries (Linux and Mac OS X)

tsuru clients are also distributed in binary version, so you can just download an executable and put them somewhere in your `PATH`.

It's important to note that all binaries are platform dependent. Currently, we provide each of them in three flavors:

1. **darwin_amd64**: This is Mac OS X, 64 bits. Make sure the command `uname -ms` prints "Darwin x86_64", otherwise this binary will not work in your system;
2. **linux_386**: This is Linux, 32 bits. Make sure the command `uname -ms` prints "Linux x86", otherwise this binary will not work in your system;
3. **linux_amd64**: This is Linux, 64 bits. Make sure the command `uname -ms` prints "Linux x86_64", otherwise this binary will not work in your system.

Below are the links to the binaries, you can just download, extract the archive and put the binary somewhere in your PATH:

darwin_amd64

- tsuru: https://s3.amazonaws.com/tsuru/dist-cmd/tsuru-darwin-amd64.tar.gz
- tsuru-admin: https://s3.amazonaws.com/tsuru/dist-cmd/tsuru-admin-darwin-amd64.tar.gz
- crane: https://s3.amazonaws.com/tsuru/dist-cmd/crane-darwin-amd64.tar.gz

linux_386

- tsuru: https://s3.amazonaws.com/tsuru/dist-cmd/tsuru-linux-386.tar.gz
- tsuru-admin: https://s3.amazonaws.com/tsuru/dist-cmd/tsuru-admin-linux-386.tar.gz
- crane: https://s3.amazonaws.com/tsuru/dist-cmd/crane-linux-386.tar.gz

linux_amd64

- tsuru: https://s3.amazonaws.com/tsuru/dist-cmd/tsuru-linux-amd64.tar.gz
- tsuru-admin: https://s3.amazonaws.com/tsuru/dist-cmd/tsuru-admin-linux-amd64.tar.gz
- crane: https://s3.amazonaws.com/tsuru/dist-cmd/crane-linux-amd64.tar.gz

Build from source (Linux and Mac OS X)

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

2.3.16 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.

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
```

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.

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".

2.3.17 Server installation guide

Dependencies

Tsuru depends on Go and libyaml.

To install Go, follow the official instructions in the language website: http://golang.org/doc/install.

To install libyaml, you can use one package manager, or download it and install it from source. To install from source, follow the instructions on PyYAML wiki: http://pyyaml.org/wiki/LibYAML.

The following instructions are system specific:

FreeBSD

```
$ cd /usr/ports/textproc/libyaml
$ make install clean
```

Mac OS X (homebrew)

```
$ brew install libyaml
```

Ubuntu

```
$ [sudo] apt-get install libyaml-dev
```

CentOS

```
$ [sudo] yum install libyaml-devel
```

Installation

After installing and configuring go, and installing libyaml, just run in your terminal:

```
$ go get github.com/globocom/tsuru/...
```

Server configuration

TODO!

2.3.18 api workflow

Tsuru sends requests to your service to:

- create a new instance of your service
- bind an app with your service
- unbind an app
- destroy an instance

Creating a new instance

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

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

Tsuru calls your service to create a new instance of your service via POST on `/resources` (please notice that tsuru does not include a trailing slash) with the "name" that represents the app name in the request body. Example of request:

```
POST /resources HTTP/1.0
Content-Length: 19

name=mysql_instance
```

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

- 201: when the instance is successfully created. You don't need to include any content in the response body.
- 500: in case of any failure in the creation process. Make sure you include an explanation for the failure in the response body.

Binding an app to a service instance

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

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

Tsuru calls your service to bind an app with a service instance via POST on `/resources/<service-name>` (please notice that tsuru does not include a trailing slash) with the "hostname" that represents the app hostname in the request body. Example of request:

```
POST /resources/mysql_instance HTTP/1.0
Content-Length: 25

hostname=myapp.myhost.com
```

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

- 201: if the app is successfully binded 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 your service does not export any environment variable, write `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_DA
```

Status codes for errors in the process:

- 404: if the service instance does not exist. You don't need to include any content in the response body.
- 412: if the service instance is still being provisioned, and not ready for binding yet. You can optionally include an explanation in the response body.
- 500: in case of any failure in the bind process. Make sure you include an explanation for the failure in the response body.

Unbind an app from a service instance

This process begins when a Tsuru customer unbinds an app from an instance of your service via command line tool:

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

Tsuru calls your service to unbind an app with a service instance via DELETE on `/resources/<service-name>/hostname/<app-hostname>` (please notice that tsuru does not include a trailing slash). Example of request:

```
DELETE /resources/mysql_instance/hostname/myapp.myhost.com HTTP/1.0
Content-Length: 0
```

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

- 200: if the app is successfully unbinded from the instance. You don't need to include any content in the response body.
- 404: if the service instance does not exist. You don't need to include any content in the response body.
- 500: in case of any failure in the unbind process. Make sure you include an explanation for the failure in the response body.

Destroying an instance

This process begins when a Tsuru customer removes an instance of your service via command line tool:

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

Tsuru calls your service to remove an instance of your service via DELETE on `/resources/<service-name>` (please notice that tsuru does not include a trailing slash). Example of request:

```
DELETE /resources/mysql_instance HTTP/1.0
Content-Length: 0
```

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

- 200: if the service is successfully destroyed. You don't need to include any content in the response body.
- 404: if the service instance does not exist. You don't need to include any content in the response body.
- 500: in case of any failure in the destroy process. Make sure you include an explanation for the failure in the response body.

Checking the status of an instance

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

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

Tsuru calls your service 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/mysql_instance/status HTTP/1.0
```

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

- 202: the instance is still being provisioned (pending). You don't need to include any content in the response body.
- 204: the instance is running and ready for connections (running). You don't need to include any content in the response body.
- 500: the instance is not running, nor ready for connections. Make sure you include the reason why the instance is not running.

Additional info about an instance

You can add additional info about instances of your service. To do it it's needed to implement the resource below:

```
GET /resources/mysql_instance HTTP/1.0
```

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

- 404: when your api doesn't have extra info about the service instance. You don't need to include any content in the response body.
- 200: when your app has an extra info about the service instance. The response body must be a JSON containing a list of fields. A field is composed by two key/value's *label* and *value*:

```
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"}]
```

2.3.19 Building your service

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.

Creating your service api

To create your service api you can use any programming language or framework. In this tutorial we will use flask.

Prerequisites

First, let's be sure 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
```

With flask installed let's create a file called api.py and add the code to create a minimal flask app:

```
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 "Hello World!".

Then, you need to implement the resources expected by the *Tsuru api workflow*.

Provisioning the resource for new instances

For new instances tsuru sends a POST to /resources with the "name" that represents the service instance name in the request body. If the service instance is successfully created, your API should return 201 in status code.

Let's create a method for this action:

```
@app.route("/resources", methods=["POST"])
def add_instance():
    return "", 201
```

Implementing the bind

In the bind action, tsuru calls your service via POST on /resources/<service_name>/ with the "app-hostname" that represents the app hostname and the "unit-hostname" that represents the unit hostname on body.

If the app is successfully binded to the instance, you should return 201 as status code with the variables to be exported in the app environment on body with the json format.

As an example, let's create a method that returns a json with a fake variable called "SOMEVAR" to be injected in the app environment. To do it in flask you need to import the jsonify method.

```
from flask import jsonify

@app.route("/resources/<name>", methods=["POST"])
def bind(name):
    out = jsonify(SOMEVAR="somevalue")
    return out, 201
```

Implementing the unbinding

In the unbind action, tsuru calls your service via DELETE on /resources/<service_name>/hostname/<unit_hostname>/.

If the app is successfully unbinded from the instance you should return 200 as status code.

Let's create a method for this action:

```
@app.route("/resources/<name>/hostname/<host>", methods=["DELETE"])
def unbind(name, host):
    return "", 200
```

Implementing the destroy service instance

In the destroy action, tsuru calls your service via DELETE on /resources/<service_name>/.

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

Let's create a method for this action:

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

Implementing the url for status checking

To check the status of an instance, tsuru uses the url `/resources/<service_name>/status`. If the instance is ok, this URL should return 204.

Let's create a function for this action:

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

The final code for our "fake api" developed in flask is:

```
from flask import Flask
from flask import jsonify

app = Flask(__name__)


@app.route("/resources/<name>", methods=["POST"])
def bind(name):
    out = jsonify(SOMEVAR="somevalue")
    return out, 201


@app.route("/resources/<name>/hostname/<host>", methods=["DELETE"])
def unbind(name, host):
    return "", 200


@app.route("/resources", methods=["POST"])
def add_instance():
    return "", 201


@app.route("/resources/<name>", methods=["DELETE"])
def remove_instance(name, host):
    return "", 200


@app.route("/resources/<name>/status", methods=["GET"])
def status(name):
    return "", 204


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

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
endpoint:
    production: production-endpoint.com
    test: test-endpoint.com:8080
```

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

Change the id and the endpoint values with the information of your service:

```
id: fakeserviceid1
endpoint:
    production: fakeserviceid1.com
```

Submiting your service

To submit your service, you can run:

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

2.3.20 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
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 create 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
```

Further instructions

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

2.3.21 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.

2.3.22 Client usage

After installing the server, build the cmd/main.go file with the name you wish, and add it to your $PATH. Here we'll call it *tsuru*. Then you must set the target with your server url, like:

Setting a target

```
$ tsuru target-add default tsuru.myhost.com
$ tsuru target-set default
```

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
```

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>
```

This 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.

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
```

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"
```

Adding hooks

By default, the commands are run from inside the app root directory, which is /home/application. If you have more complicated deploy related commands, you should use the app.yaml pre-restart and post-restart scripts, these will run before and after the restart of your app, which is triggered everytime you push code or call restart. Below is an app.yaml sample:

```
hooks:
  pre-restart:
    - deploy/pre.sh
  post-restart:
    - deploy/pos.sh
```

You should put app.yaml file in the root directory of the app, and scripts are relative to it (you can use absolute path for scripts too, for instance `/usr/bin/bash`).

Further instructions

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

2.3.23 Deploying Python applications in tsuru

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.

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 1 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    |
+-------------+-------------------------+-------------+--------+
```

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/globocom/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

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: 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@cloud.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@cloud.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@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: 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.

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@cloud.tsuru.io:blog.git
   a211fba..bbf5b53  master -> master
```

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 that 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@cloud.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@cloud.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/.

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 none 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 variable 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@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 `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`.

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 `app.yaml` and runs `pre-restart` and `post-restart` hooks. As the extension suggests, this is a YAML file, that contains a list of commands that should run in pre-restart and post-restart hooks. Here is our example of app.yaml:

```
hooks:
  post-restart:
    - python manage.py syncdb --noinput
    - python manage.py migrate
```

It should be located in the root of the project. Let's commit and deploy it:

```
$ git add app.yaml
$ git commit -m "app.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 post-restart
remote:
remote:  ---> Deploy done!
remote:
To git@cloud.tsuru.io:blog.git
   a780de9..1b675b8  master -> master
```

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

Going further

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

2.3.24 Deploying Ruby applications in tsuru

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.

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 1 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    |
+-------------+------------------------+-------------+--------+
```

Application code

This document will not focus on how to write a Rails blog, you can clone the entire source direct from GitHub: https://github.com/globocom/tsuru-rails-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`)

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: 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@cloud.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@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.

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 `Gemfile` 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-ruby??` package using `gem`, and this package depends on two `apt-get` packages: `ruby-dev??` and `libmysqlclient-dev`, so here is how `requirements.apt` looks like:

```
libmysqlclient-dev
ruby-dev
```

And here is `Gemfile`:

```
...
```

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@cloud.tsuru.io:blog.git
   a211fba..bbf5b53  master -> master
```

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

Now that 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@cloud.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 `Gemfile` file:

```
...
^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@cloud.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/.

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 none 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 variable 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@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 `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`.

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 `app.yaml` and runs `pre-restart` and `post-restart` hooks. As the extension suggests, this is a YAML file, that contains a list of commands that should run in pre-restart and post-restart hooks. Here is our example of app.yaml:

```
hooks:
  post-restart:
    - python manage.py syncdb --noinput
    - python manage.py migrate
```

It should be located in the root of the project. Let's commit and deploy it:

```
$ git add app.yaml
$ git commit -m "app.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 post-restart
remote:
remote:  ---> Deploy done!
remote:
To git@cloud.tsuru.io:blog.git
   a780de9..1b675b8  master -> master
```

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

Going further

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