Pitaya-Bot Documentation

TFGCo

Sep 02, 2019

Contents:

CHAPTER 1

Overview

Pitaya-Bot is an easy to use, fast and lightweight test server framework for Pitaya. The goal of pitaya-bot is to provide a basic development framework for testing pitaya servers via integration tests or stress tests.

1.1 Features

- **No code writing** - Pitaya-Bot only needs JSON specs and a configuration YAML, in order to work. It is simple to create and test directly into any environment, be it development or production.
- **Concurrency** - Configurable number of instances, which will run the tests.
- **Monitoring** - Pitaya-Bot is configurable to work with Prometeus. It allows the user to see metrics of the server, which the tests are being run. This way, it is possible to run stress or integration tests.
- **Communication** - Communication between server and client enabled for TCP via JSON.
- **Handler Support** - Support handler messages, to emulate the behaviour of customers using Pitaya.
- **Summary** - At the end of the tests, returns if the requests made have the expected responses. Perfect for testing idempotence of the server.

1.2 Who's Using it

Well, right now, only us at TFG Co, are using it, but it would be great to get a community around the project. Hope to hear from you guys soon!

1.3 How To Contribute?

Just the usual: Fork, Hack, Pull Request. Rinse and Repeat. Also don't forget to include tests and docs (we are very fond of both).

CHAPTER 2

Features

Pitaya-Bot has been developed in conjunction with Pitaya, to allow the usage of every feature contained in Pitaya, inside this testing framework. It has been created to fulfill every possible testing scenario and make it easy to be used, without the need to write code.

Some of its core features are described below.

2.1 No code writing

The tests which will be run don't need the knowledge of Golang. The writting of JSON specs and configuration are more than enough.

2.2 Handler Support

It is only possible to test handlers, due to the fact that this framework is focused on the scenarios which the user takes part.

The tests can be created to test idempotency or stress the server and see how it behaves.

2.3 Bots

Bots are "fake" users, which will be doing requests to Pitaya servers. All of them must implement the Bot interface.

Pitaya-Bot comes with a few implemented bots, and more can be implemented as needed. The current existing bots are:

2.3.1 Sequential

This bot follows exactly the orders written inside the JSON spec and chronologically, one bot after another in each instance.

2.4 Concurrency

In the test setup, it is possible to inform the number of instances that will be doing it. So that it is possible not only to make integration tests, but also stress tests.

2.5 Monitoring

Pitaya-Bot is configurable to measure the server health via Prometeus. It is perfect for the testing, because the tester will be able to see how the server behaves with any number of requests and any handler that he wants to test.

2.6 Storage

Storage is the space that the Bot will retain the information received from Pitaya servers, so that it can be used in future use cases. All of them must implement the Storage interface. The desired storage must be set via configuration and will be created via factory method `NewStorage`. Remember to add new storages into this factory.

Pitaya-Bot comes with a few implemented storages, and more can be implemented as needed. The current existing storages are:

2.6.1 Memory

This storage retains all information inside the testing machine memory. The stored information is not persistent and will be flushed with the end of the test.

2.7 Custom initialization and wrap-up

Specs can specify custom initialization and wrap-up routines to do operations such as fetching an initial state from some storage and saving the final state to a storage.

To define an initialization function in the script you should create a *preRun* field, with *function* specifying which function should be run. It also accepts *args* as an object with arguments to be passed to the function.

To define a wrap-up function in the script you should create a *postRun* field, with *function* specifying which function should be run. It also accepts *args* as an object with arguments to be passed to the function.

The JSON testing sample has an example with these fields.

2.7.1 Redis

These initialization and wrap-routines run lua scripts in redis and come with default scripts to fetch a state from a set and save it to another. The preRun script is expected to return the initial state for the bot and the postRun script receives the final state and is expected to do something with it.

The default initialization script tries to fetch an element from the set *${name}:available* and write it to *{name}:used.*

The default wrap-up script writes the state to *${name}:available*.

The initialization script accepts two arguments:

- **name (required)**: the key argument that is passed to the lua script
- **failEmpty (optional)**: a boolean indicating if the method should fail if the script returns nil

The wrap-up script accepts one argument:

- **name (required)**: the key argument that is passed to the lua script

2.8 Serializers

Pitaya-Bot supports both JSON and Protobuf serializers out of the box for the messages sent to and from the client, the default serializer is JSON.

2.9 Spec generation

It is possible to create specs from pitaya-cli history by using the `parseHistory` command.

CHAPTER 3

Workflow

In this section we will describe in details the available workflow processes, since the setup until the end summary. The following examples are going to assume the usage of a sequential bot with TCP communication and JSON information format.

3.1 Basic Workflow

The overview of what happens when pitaya-bot is started:

- Initialization of app, configuration fetch, specs directory lookup and creation of metric reporters
- Instantiation of many go routines, which are defined in spec files
- Validation of selected bot and written specs
- Execution of specs
- Notification of the result to all of metrics reporter
- Summarization of tests

3.1.1 Initialization

The first thing pitaya-bot does is instantiate an App struct based on the config file, receiving the metric reporters that will be used (Promethues, . . .) and name of the pitaya game which will be tested.

The configuration is also passed to the bots that will follow the specs, so that they know which storage will be used, endpoint to access, etc.

Another important point is the directory where the specs are located, because it will use the number of spec files as the number of go routines that will execute each one of them in a parallel way.

3.1.2 Instances

Based on the spec file, the field `numberOfInstances` will dictate how many go routines will be created to run each of the written scenarios.

3.1.3 Validation

For each spec, it will validate if it was able to:

- Create the given type of bot
- Initialize the bot
- Run the given spec without problems
- Finalize the bot

3.1.4 Execution of given spec

In the moment that the bot is initialized, it will fetch all the information contained in the spec and create operations that will be executed. The operations can vary, it can make all the possible operations that a pitaya client can do and also store informations from the received responses. It is important to mention that each bot has different operations that can be used, so consult them before writing your own testing scenarios.

3.1.5 Metric Reporter

After each request to a pitaya server, the pitaya-bot will inform the metric reporter of the response time, which is important to see the overall QoS(Quality of Service).

3.1.6 Summary

After all specs have been run, it will gather all the results obtained and return in the terminal, informing if it was a total success or if some errors occurred.

3.2 Workflows

There is the listing of all possible workflows:

1. *Local*: Pitaya-bot will be instantiated locally and will request the server from current location
2. *Local Manager*: A Pitaya-Bot manager will be instantiated locally and will create the Kubernetes Jobs inside kubernetes cluster from given config and specs
3. *Remote Manager*: A Pitaya-Bot manager will be instantiated inside a kubernetes cluster and will create the Kubernetes Jobs from given config and specs

3.2.1 Local

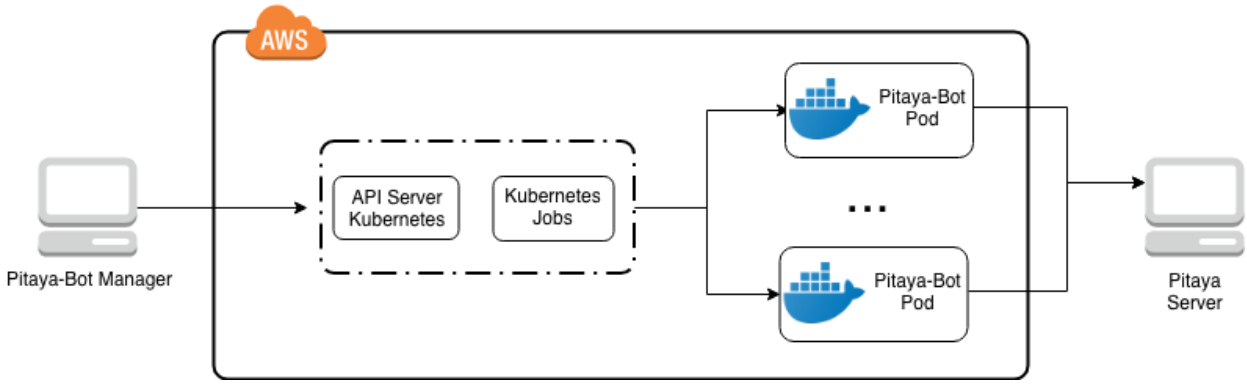
It will instantiate an unit of pitaya-bot, which will run all specs located inside given directory. Each spec file will be run in a distinct go routine and also, each operation from the spec will be run in another distinct go routine.

The local architecture is represented below:

3.2.2 Local Manager

It will instantiate a pitaya-bot manager, which will create all configmaps, containing all specs and the config.yaml, to be used by each kubernetes job. After creating all configmaps and jobs, it will start a controller, that will be watching all the jobs created and after all of them finish their work or time out, it will clean everything that was created inside the kubernetes cluster.

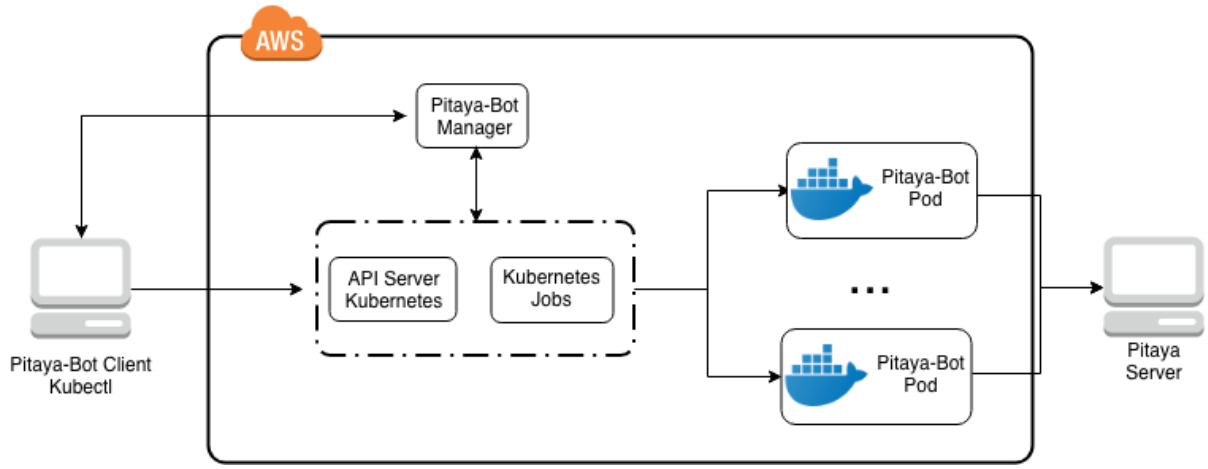
The local manager architecture is represented below:



3.2.3 Remote Manager

It will instantiate a pitaya-bot manager inside kubernetes cluster, which will create all configmaps, containing all specs and config.yaml, to be used by each kubernetes job. After creating all configmaps and jobs, it will start a controller, that will be watching all the jobs created and after all of them finish their work or time out, it will clean everything that was created and will also delete itself with the its configmaps.

The remote manager architecture is represented below:



3.2.4 Deploy Manager

It will create a kubernetes deployment, which will be running a pitaya-bot remote manager inside a kubernetes cluster.

3.2.5 Delete All

It will delete everything that is related to pitaya-bot inside the kubernetes cluster and is mentioned in the config.yaml.

CHAPTER 4

Configuration

Pitaya-Bot uses Viper to control its configuration. Below we describe the configuration variables split by topic. We judge the default values are good for most cases, but might need to be changed for some use cases. The default directory for the config file is: ./config/config.yaml.

4.1 General

These are general configurations

Configuration	Default value	Type	Description
game		string	Name of the application being tested, to appear in Prometheus

4.2 Prometheus

These configuration values configure the Prometheus monitoring service to check how the server being tested is behaving. To monitor the application, the option *report-metrics* must be true when starting the Pitaya-Bot.

Configuration	Default value	Type	Description
prometheus.port	9191	int	Port which the Prometheus instance will run

4.3 Server

The configurations needed to access the Pitaya server being tested

Configuration	Default value	Type	Description
server.host	localhost	string	Pitaya server host
server.tls	false	bool	Boolean to enable/disable TLS to connect with Pitaya server
server.serializer	json	string	must be json or protobuffer
server.protobuffer.docs	""	string	Route for server documentation. Target server must implement handlers for protobuf descriptors and auto documentation.
server.protobuffer.pushinfo.routes	[]	string	Information about the protos used by push messages from the server, this part contains the routes of the messages
server.protobuffer.pushinfo.protos	[]	string	Information about the protos used by push messages from the server, this part contains the names of the protos

If your application use protobuffers, specifying docs is required. You can also add a list of routes and protobuffer types if your application sends push information to the bot. See `testing/protobuffer/config/config.yaml` for example.

4.4 Storage

Configuration	Default value	Type	Description
storage.type	memory	string	Type of storage which the bot will use

4.5 Kubernetes

Configuration	Default value	Type	Description
kubernetes.config	$HOME/.kube/config	string	Path where kubernetes configuration file is located
kubernetes.context		string	Kubernetes configuration file context
kubernetes.cpu	250m	string	CPU which will be allocated for each Kubernetes Pod
kubernetes.image	tfgco/pitaya-bot:latest	string	Pitaya-Bot docker image that kubernetes will use to deploy pods
kubernetes.imagepull	Always	string	Kubernetes docker image pull policy
kubernetes.masterurl		string	Master URL for Kubernetes
kubernetes.memory	256Mi	string	RAM Memory which will be allocated for each Kubernetes Pod
kubernetes.namespace	default	string	Kubernetes namespace that will be used to deploy the jobs
kubernetes.job.retry	0	int	Backoff limit from the jobs that will run each spec file

4.6 Manager

Configuration	Default value	Type	Description
manager.maxrequeues	5	int	Maximum number of requeues that will be done, if some error occurs while processing a job
manager.wait	1s	time.Period	Waiting time between each job process

4.7 Bot

Configuration	Default value	Type	Description
bot.operation.maxSleep	500ms	time.Duration	Maximum sleep duration between bot operations, the launcher selects a random value in the range [0, maxSleep]
bot.operation.stopOnError	false	bool	Defines if the bot should stop running on error, by default it restarts the spec
bot.spec.parallelism	1	int	Defines the number of instances to run for each spec when running on kubernetes

4.8 Custom initialization and wrap-up

Configuration	Default value	Type	Description
custom.redis.pre.url	redis://localhost:6010	string	Redis url to connect if using a custom redis initialization
custom.redis.pre.connectionTimeout	10	int	Timeout in seconds to connect to redis
custom.redis.pre.script	""	string	Path to the lua script to run if using a custom redis initialization
custom.redis.post.url	redis://localhost:6010	string	Redis url to connect if using a custom redis wrap-up
custom.redis.post.connectionTimeout	10	int	Timeout in seconds to connect to redis
custom.redis.post.script	""	string	Path to the lua script to run if using a custom redis wrap-up

CHAPTER 5

Command Options

Pitaya-Bot is a CLI application, that has many command options, which will be described below by topic. We judge the default values are good for most cases, but might need to be changed for some use cases. The default verbosity for the application logger is Debug.

5.1 Pitaya-Bot

Base configuration needed to run pitaya-bot

Command	Command Letter	Default value	Type	Description
config		./config/config.yaml	string	Config file path from pitaya-bot
dir	d	./specs/	string	Specs directory
duration		1m	time.Duration	Minimum total duration of tests
report-metrics		false	bool	Enable/Disable metrics reporter
pitaya-bot-type	t	local	string	Pitaya-Bot workflow type that will be executed. It can be: local, local-manager, remote-manager, deploy-manager, delete-all
delete		false	bool	Delete all pods, config maps, jobs and deployements before run. Only available when pitaya-bot-type is local-manager or remote-manager.

5.2 Logger

These are logging configurations

Command	Command Letter	Default value	Type	Description
verbose	v	3	int	Logger verbosity level => v0: Error, v1=Warning, v2=Info, v3=Debug
logJSON	j	false	bool	Enable/Disable logJSON output mode

CHAPTER 6

Test Writing

6.1 Command Options

The execution of pitaya-bot offers many command options, that enable/disable many functionalities and different types of workflows. The command options can be found in the command section.

6.2 Configuration

It is important to create the config.yaml file before running the tests, so that pitaya-bot knows which server to access and which report metrics to use. The configuration options can be found in the configuration section.

6.3 Spec Configuration

Before executing any spec, it is possible to use the following options:

- `numberOfInstances`: The number of instances(go routines) that will run the same spec in parallel

6.4 Bots

There are bots that will be able to follow the operations given to them in each spec file. The available bots are:

6.4.1 Sequential Bot

This bot will follow the orders contained inside a spec file sequentially and chronologically. The possible operation types for it are:

- `Request`: Requests pitaya server being tested

- `Notify`: Notifies pitaya server being tested
- `Function`: Internal operations for the bot, such as:
  - `Disconnect`: Disconnect from pitaya server
  - `Connect`: Connect to pitaya server
  - `Reconnect`: Reconnects to pitaya server
- `Listen`: Listen to push notifications from pitaya server

6.5 Operation

Operation is the generalistic struct which contains the action that the specified bot will do. The fields are:

- `Type`: Type of operation which the bot will do. Each bot has different types
- `Timeout`: Time that the bot has to execute given operation
- `Uri`: URI which the bot will use to make request, notification, listen, ...
- `Args`: Arguments that will be used in given operation
- `Expect`: Expected result from operation
- `Store`: Which field from the response it should retain

6.6 Special Fields

These are fields that when used will fetch the information from given structure:

- `$response`: When used in `Expect` field as key, will get the object response, that can access his attributes via `.` or `[]`
- `$store`: The information contained inside a storage, can be used as a `Expect` value or `Args` value.

6.6.1 Config example

Below is a simple example of a config file, for another one which is being used, check: config

```
game: "example"

storage:
  type: "memory"

server:
  host: "localhost",
  tls: true

prometheus:
  port: 9191
```

6.6.2 Spec example

Below is a base example of a spec file, for a working example, check: spec

```
{
  "numberOfInstances": 1,
  "sequentialOperations": [
  {
    "type": "request",
    "uri": "connector.gameHandler.create",
    "expect": {
      "$response.code": {
        "type": "string",
        "value": "200"
      }
    },
    "store": {
      "playerAccessToken": {
        "type": "string",
        "value": "$response.token"
      }
    }
  }
  ]
}
```

6.6.3 Testing example

For a complete working example, check the testing example.

CHAPTER 7

Examples

Example projects can be found here

P.S. If you are using Minikube and have a testing pitaya server running locally(host OS), change the config.yaml to an address that the pods from minikube can access. Normally it is: **192.168.99.1**, but you can find it running **ifconfig** from host OS and looking at the inet from **vboxnet**

CHAPTER 8

Indices and tables

- genindex
- modindex
- search