Instructions for examiner

This project is up and running here notice that there is no route handler for '/' so it will just return a 404 since there was nothing about this in the requirements.

To get all containers up and running you should start with

```
curl -X PUT localhost:8081/state?payload=INIT
```

After this you can try other commands that are outlined under running the project.

When you are done, please remember to shut down the system with

```
curl -X PUT localhost:8081/state?payload=SHUTDOWN
```

A list of implemented features can be found discussed here

Description

Original project outline can be found in the attached assignment.pdf document

The goal of this solo-developed project is to implement a service which utilizes RabbitMQ to communicate messages between worker threads with two REST API ends points which control the state of the system and return information from it. The application is running on Heroku and is backed by a CI/CD pipeline running on a local instance of Gitlab.

This project is up and running here notice that there is no route handler for '/' so it will just return a 404. Read further

Component-wise breakdown:

ORIG

- Sends messages on a loop 3 seconds apart in topic my.o
- Possible messages sent by the service include
    - MSG_1
    - MSG_2
    - MSG_3
- Service can be in states
    - PAUSED - messages stop sending
    - RUNNING - messages start sending again
    - SHUTDOWN - in dev stops sending messages, in production container is scaled down
    - INIT - container is starting up, will start delivering messages when started. State automatically set to RUNNING

IMED(Intermediate)

- Subscribes for messages from my.o (sent by orig)
- Publishes message to my.i
- Possible message combinations include (depending on orig input)

    - Got MSG_1
    - Got MSG_2
    - Got MSG_3
- Service can be in states
    - SHUTDOWN - messages are no longer received
    - INIT - messages can be received again

OBSE(Observer)

- Subscribes for all messages within the network, therefore receiving from both my.o and my.i
- Stores the messages into a file
- Examples of possible messages saved by the service include
    - 2020-11-01T06:35:01.373Z MSG_1
    - 2020-11-01T06:36:02.373Z GOT MSG_1
        - Timestamps are ISO typestamps
        - All IMED and ORIG message types are covered in similar fashion
- Service can be in states
    - SHUTDOWN - messages are no longer received
    - INIT - messages can be received again

HTTPSERV

- When requested, returns content of the file created by OBSE
- Service can be in states
    - SHUTDOWN - REST API is not responsive
    - INIT - REST API is up
- On local development the service is accessible via `http://localhost:8082`
    - 8082 was set as the port so as to not conflict with local Gitlab runtime

APIGATEWAY

- Implements an API service to control IMED, ORIG and OBSE workers and HTTPSERV api
- Communicates with ORIG via special control messages
- Service can be only in RUNNING state, because otherwise it could not INIT other services. In effect it is always on and serves as the main gateway to the service
    - RUNNING - can send commands to other services and query them

Gitlab

- The repository includes a local Gitlab runtime, which can be found outside vh4 folder in root `./gitlab-ci`
- You can start the service with `docker-compose up --build` in it's folder
- You can register a runtime for test execution by running `gitlab-runner-register.sh` executable
- Information on how to setup multiple git remotes can be found in section Working with multiple git remotes
- Adding authentication and working with git in general is assumed to be common knowledge and not outlined here
- The deployment and testing procedures are outlined in repository root at `gitlab-ci.yml`

- In order to deploy to Heroku you need to define `HEROKU_API_KEY` in the Gitlab CI/CD settings and this key should be requested separately from the maintainer

Implemented project requirements

Requirements are outlined in the assignment document and project root

- [x] Install the pipeline infrastructure using gitlab-ci
- [x] Create, setup and test an automatic testing framework
- [x] Implements changes to the system by using the pipeline. The development should be done in test-driven manner. Further notes on this in section Testing
- [x] implement a static analysis step in the pipeline by using tools like jlint, pylint or SonarQube. This is implemented with Eslint in repository root.
- [x] Deploy the application at least to your own machine. Done via deployment to Heroku container service, outlined in detail in section Deployment
- [ ] Implement monitoring and logging for troubleshooting. Not implemented because of time limitations. However services log to console which is accessible via `heroku logs --tail -a application-name`
- [x] Provide an end report. This is available in repository root (not project root) with the name `EndReport.pdf`, which is effectively an export of this Markdown file.

Getting started

Requirements

- Docker runtime
- Node.js
- Heroku CLI if you want to deploy manually and not through the CI/CD pipeline. Note that this requires authentication that can be requested from the maintainer
- Yarn
- A browser, Curl, Postman or similar for communicating with the service
- An internet connection 😃

Running the project

```
docker-compose up --build
```

Once the service is healthy, it can start receiving messages. Once apigateway is up, methods can be called. The service will automatically be set to RUNNING mode and messages will be sent.

You can put the service in SHUTDOWN state with `curl -X PUT localhost:8081/state?payload=SHUTDOWN`

You can put the service in INIT state with `curl -X PUT localhost:8081/state?payload=INIT`

You can put the service in PAUSE state with `curl -X PUT localhost:8081/state?payload=PAUSE`

You can put the service in RUNNING state with `curl -X PUT localhost:8081/state?payload=SHUTDOWN`

You can get the state of the service with `curl localhost:8081/state`

You can get a timestamped log of state changes with `curl localhost:8081/run-log`

You can get a log of messages sent (and stored by obse) with `curl localhost:8081/messages`

Note that SHUTDOWN will clear messages sent and stored by the service!

Project structure

The project is split into self contained micro services with their of package.json and dependency management. Each is it's own container and deployed to a separate Heroku instance. Local development is orchestrated via docker-compose

This way they can change independently from each other. This is a bit inconvenient because we need to run yarn or npm install separately in each package. However this is not needed when development happens in the Docker environment.

- project root includes assigment.pdf which includes detailed instructions on the project and our docker-compose config file
- Each aforementioned module is listed under source with their own package.json, env, README and so on
- static code analysis is done via Eslint which is shared from repository root

In addition to services mentioned in the description, there are two additional utility packages:

rabbitmq-helpers

- Implements a queue wrapper to send and consume both fanout and topic based messages
- This package is found in `./src/rabbitmq` and is published as a public npm module
- You can create updates to the service in the folder and distribute them via `npm publish`. In versioning we follow the Semver convention.

s3

- Implements methods to upload, get and delete files from S3
- This service is circumvented in local and local docker development via using the filesystem to remove unnecessary network requests
- As of writing the service does not include unit tests but it has been tested extensively in practice. Nevertheless unit tests should be added in the future.

Development

Non Docker Development

If you want to develop individual serices, you need to initialize rabbitmq service separately with `docker run -d -p 15672:15672 -p 5672:5672 rabbitmq:3-management` after which you can access it on the browser at `localhost:15672` with password and username = guest

You should be able to run each service via it's start script after a yarn install and have them communicate with each other. However a combined startup is not orchestrated yet, so a recommended method would be

to use our docker-compose for development

Docker based development

As mentioned earlier you can run the project with `docker-compose up --build` in project root (project, not repository). This will spin up all necessary services and set them in RUNNING mode.

A big gotcha at the time of writing is that there is no code hot-reload into the container, so in order for your changes to reflect to the runtime you need to do another docker build. Current workflow around this is to develop the services separately outside docker environment and then integration test them with docker-compose

Testing

Each package uses mocha as it's test runner sinon for test spies, stubs and mock and expect for test result assertion. In some cases sinon assertions are used such as for spies. For test code coverage we use Istanbul Reports. Tests were placed in each package separately for easier test management and separation of concerns. This is against requirement 3 `Tests mush be in a separate folder “tests” at the root of your folder tree.` but the decision was made for ease of development and because it will not affect any functional requirements. This can be updated separately if this is a hard requirement, but it will make path resolution more confusing and make test discoverability harder.

The main focus of this project so far has been to have wide coverage on unit tests, which are writted for each module following the Test-driven Development principles. Additional integration tests could, and should, be added in the future. However this was removed as a goal because of time limitations, but will be improved upon later.

CI/CD pipeline

This project uses a local Gitlab instance with a integrated test runner. Tests are run on push on each branch but deployments happens only on master branch pushes. A preferred mechanism would be to use a PR setup with reviews but this was not implemented because I worked as a single developer directly on master to speed up development. The configuration of the pipeline is contained in `.gitlab-ci.yml` in the repository root. The configuration had to be put outside the project root in order for gitlab to detect and run the job.

Each package has it's own tests which are set close to the code to be tested as discussed in section testing. Tests use mocha as test runner sinon for test spies, stubs and mock and expect for test result assertion. However some assertion is done also via Sinon such as in cases where I used test spies. For test code coverage we use Istanbul Reports. If a test fails in the pipeline, the job fails and the deployment script does not run. Test results can then be accessed in the jon log on Gitlab jobs panel. If the tests pass a deployment is started automatically.

Deployment is done via Heroku CLI which is installed as a beforescript in the test runner. Downside here is that we need to install the CLI and also Node in order to run it and use a docker image. This makes the image quite heavy, but on another note a test run in the pipeline takes on average 3 minutes, which I believe is acceptable. Another way to go about shaving this time down would be to use a smaller Docker image and deploy directly to Heroku's docker registry and use the Heroku's API to release new versions.

However the method I used saved precious development time and was acceptable as it was. Because of limitations of how Heroku handles multi-container deployments some rather ugly "cd" operations have to be done to deploy the right things. This, in addition to other Heroku related headaches, is discussed more in detail in section deployment

Example runs

Example CI/CD runs of passes and fails in the CI/CD pipeline can be found in the folder `ci-cd-example-runs` in project root.

Deployment

This service is deployed to Heroku automatically in our Gitlab CI/CD pipeline using separate containers for each service and using their CLI for deployment.

Heroku was chosen because of it's relatively easy setup and a free tier that just supports the 5 apps required to run this project. However there are special considerations which are not outlined in Heroku's documentation clearly

First of all we are dealing with a multi container deployment which is more difficult than the basic one container deployment which is outlined in most guidelines. For instance in Heroku's documentation this is only briefly discussed and discussed a bit more in detail in this article

The topics that are left out in the documentation include:

1. Your app doesn't necessarily have an instance running automatically and you need to run: `heroku ps:scale your-instance=1 -a your-app-name` to get it running
2. The container app can only have one entrypoint (web) and everything else has to be a worker instance. Because of this I had to separate httpserv and apigateway to run on their own instances.
3. On free tier you can have only one containers running on one instance
4. In order for the entrypoint to work, you need to define it as service type web, and in the multi-container mode you need to name the Dockerfile as Dockerfile.web. In the beginning I was using Dockerfile.httpserv and then wondered why I was getting error `H14 error in heroku - “no web processes running”`, mind you, this same error happens also if you have no instances running as outlined in point 1. However because we have two web instances, the recursive deployment from project root doesn't work and we have to cd into each sub folder and do a publish and release there against a normal Dockerfile

Details on the process model [here](heroku container:push web -a devops-apigateway)

Steps of deploying to Heroku container service for multiple images in recursive mode

1. You need an app to run against `heroku create`
2. You need some images, they needs to be nested in sub directories as Dockerfile.servicename notice that the service managing HTTP calls needs to be called Dockerfile.web to work
3. You push images to heroku with `heroku container:push --recursive -a your-created-app-name`, this will build the images a push them

4. You can always push new changes after doing changes to the files, if there are no changes no new image is uploaded
5. You can also defined specifically what you want to push as `heroku container:push --recursive orig imed -a devops-imed-orig` which will push Dockerfile.imed and Dockerfile.orig
6. You can then release containers with `heroku container:release web otherservice other2service -a your-created-app`

Learnings and reflection

In this section I discuss each part of the implementation and why I made the choices I did in addition to general reflection on the assignment. A general note on implementation requirements was that the containers were required to be shut down on SHUTDOWN message. However I decided partly against this for a couple of reasons.

1. The local development environment doesn't need to scale containers up and down and just simulating shutdown and init behavior gets the job done. Then in production we can separately call each container to be actually shutdown and setup when needed.
2. The wording was to 'kill all containers' which sounds that apigateway should be part of this. I decided against this because this is the main point of contact to the service and if SHUTDOWN would bring it also down and INIT functionality would be effectively meaningless
3. Each container both in development and production will receive and confirm SHUTDOWN and INIT messages to apigateway before the state change request resolves. This way we can have a controlled shutdown of our service and not just kill it and also we get the benefit of waiting anc confirming on the request whether or not the services were shut down correctly.

API Gateway

I'm generally happy with how this service turned out. The routing and controller structure is clear with tests placed near to what they are testing for easy updatability and discoverability. For added abstraction routes could be separated into a separate file, but this was not done because the routes were quite simple.

The biggest headache of this was to implement the state controller. Firstly the state itself could work better as a class since it has a state (as the name implies) and would make the implementation cleaner. I was planning refactoring this but ran out out time. In addition there were some edge cases that were discovered late in development, for instance that the topic based messages such as RUNNING and PAUSE don't include accurate (to the millisecond) timestamps. This is due to the complexity of setting up rabbitmq in a reliable way and how I iterated many times over on the philosophy on how communication happens between services. This is discussed more in detail in section Rabbitmq

In general what would have remedied this issue would have been to spend more time planning rabbitmq messaging between services. This integration point was implemented and tested as the last step in the process and with limited implementation time and without proper integration tests a big bang approach had the consequences that could be expected. However on another note adding those tests in place would have increased the workload with a rough estimate of 10-15 hours which would've sent the total time spent on the project to around 80 hours.

One regret is also the state of documentation. I wanted to use swagger with it's yml based route definitions, a friendly ui for documentation, testing and development, but I ran out of time in the project. This would definitely be one of the first things that I would add to the project because it makes understanding and working with the implementation so much easier.

HTTPSERV

This service is a rather barebones express server and was kept that way. Implementation did not change excluding headaches around CORS configuration which was remembered late in development and added confusion when testing the `/messages` route especially in Heroku environment. This would have been picked up and fixed earlier if development had been done natively in Docker environment.

IMED

This service was changed into a class based implementation to clean up the code and manage queue connections better. No special considerations here, but suffered from issues discussed in section Rabbitmq

OBSE

This service was changed into a class based implementation to clean up the code and manage queue connections better. The debatable choice made here was to use S3 as the backbone of storing messages. This is not the optimum choice because S3 is an object storage, whereas the appending nature of the operation would call for a filesystem, or block based storage. This lead to an awkward configuration where on append we first get the last log from s3, append to it and then upload it back. In practice this is not an issue in the scope of this project, but if the service was to run long periods of time even the requested small text file would ultimately resolve slowly enough to cause performance issues. The decision of using S3 was made with time constraints in mind because it is easy to setup and use. The method of storing and getting messages was abstracted into their own `deletefile` and `appendToFile` methods so that this component could be easily changed in the future which I definitely would do given more time.

ORIG

This was the first worker service that I realized a class based implementation would be easier to manage, and hence I spent majority of my time with this and implementing queue wrappers around it. An interesting learning here was that whereas in browser Javascript a setInterval returns an id, with which the interval can then be cleared with clearInterval(id) and the interval will not be defined anymore. However in Node.js setInterval returns an object, which changes implementation details.

RabbitMq

This was the major headache of the setup. Firstly from the first implementation in project `vh3` I would always just restart the containers when there were errors in RabbitMQ connection. However since apigateway should always be on and robust to errors, this was not an option. This lead me down a rabbit hole of how to manage situations where the rabbitMQ is not set up yet and the connection fails and also how to then manage an incoming message when the connection is not up without loosing that message. In addition there was not a lot of documentation around this at least in Node.js and my final solution was done via experimentation and intensive googling. However I'm rather pleased with how the code turned out where we continually retry connections without affecting the experience on the API side. Any message

sent before RabbitMQ is healthy, or if it goes down during runtime, is promise based and will be send once the connection is up and running again.

Second headache was how to manage communication between services. Since I decided on that SHUTDOWN and INIT should be received by all the services which calls for a fanout queue, but as per requirements orig handles only PAUSE and RUNNING states, which calls out for a topic based queue, I had to find a way to separate these. In addition we should also support the other my.x type topic messages.

I sought out to then have two types of messages, control messages that control the state of the service that are sent via a fanout queue to all services, these included SHUTDOWN and INIT. Then a topic based control queue separately between apigateway and orig to pass RUNNING and PAUSE messages. Lastly I would have a topic queue for my.x type messages. These would be governed by two exchanges, one for fanout and one for topic.

However I made a mistake in the implementation. The topic messages had a structure of

```
{
  message:'MSG_1' // notice missing timestamp, we'll get back to this later
}
```

and topic would be read from the routingKey.

The fanout type messages had a type of

```
{
  payload:'INIT',
  id:123124124,
  timestamp:12313123
}
```

however when I created a message handler for my queue, I failed to realize that I structured PAUSE and RUNNING messages in the fanout format, but sent them via a topic based queue. This would break down the logic and I had to add a patch late in development to work around this. In addition because I was sending PAUSE and RUNNING messages over the topic queue, I would not have timestamps for these which was required to store timestamped state changes. To work around this I added a timestamp to the message when the apigateway got a response from orig.

This is another thing that should be mentioned. In order for me to have flow of:

1. REST api receives change state request
2. State controller sends message to required services
3. State controller waits for response from services before resolving the request

I had to find a way to know when the response arrives from the service. I couldn't use the typical way of consuming messages from the queue directly, because this would break the abstraction between queue and state, and I also didn't want to add special callbacks because this would've made the implementation even more complex. What I then decided to do was to use a polling mechanism, where when the REST api

gets a state change request it would be assigned an id. This message would go to the service and on completion it would send the same message back which we would find by id. If something went wrong the message would also contain an error so we can send that back to the user. An error case example is when you try to change state to `RUNNING` when the current state is `SHUTDOWN`. This polling mechanism worked okay and is relatively clear in it's implementation in apigateway state controller, however what is missing is a timeout. If for some reason for instance the orig service is not up and we change the state to RUNNING, the request will hang until it times out. This is not a very elegant way to handle the issue, but on the other hand if the REST call times out we can always try again because the local state doesn't update before the service resolves and we can always send the request again.

Another headache was figuring out how to mock the queue system in unit tests. This is because I did not want to initialize a RabbitMQ instance just to pass the tests, so I ended up implementing a mock class that can be imported from the module. This way we can then selectively initialize either the mock queue or the actual queue instance depending on the environment. This was not the most elegant solution, but figuring out how to mock the queue when it was sometimes two layers of abstraction down proved to be a time sink from which I decided to get away to get the project done.

In general I think the queue system was the best way to go given constraints by Heroku, that is that I had only 5 free instances to use. I could've had one each worker have a separate API as a web instance that passes messages to its worker, however it was not clear would a worker instance with a web instance in a dyno count as one or two instances thus breaking the free limit and also how would that communication between the web and worker instances happen. In addition using the queue gave me an opportunity to delve deeper into RabbitMQ whilst making the service also more robust because a http call can fail and to work around that I'd have to create a logic of retrying requests and handling edge cases.

Heroku

Heroku had some major headaches to get to behave with a multi-container setup discussed more in depth in section deployment. In addition to this I had to work around free tier limitations as I'm not willing to spend money on this project, only precious time. However once it was setup everything worked quite well. I could have an rabbitMQ instance as an addon and just a couple of env configs later things would be up and running.

If I was to do this project again, and had more time and money, I would look into using a swarm deployment, use AWS container service with Elastic Beanstalk or maybe even venture out and setup a Kubernetes cluster each of these options offering better scaling abilities than what Heroku has to offer. This is even more true because the current implementation only scales between 1 and 0 instances and doesn't then scale to demand, however this was not outlined as a requirement hence I didn't consider the issue further.

General thoughts and feedback

This was a huge project. Hands down I did not expect the amount of thinking and work I put into this. I did end up adding additional features like the static code analysis and hosting the service online (which itself took me at least 15 hours of work), and maybe in retrospect I should've focused more on the bare necessities as a one developer team. Even with this I'm pleased with the end result however feeling the pain of unpolished work.

I'm not sure what is the expectation on the talents of a student to complete this assignment, but at least as a single developer just getting the bare minimum functionality seems like a high call. When this is combined with the requirement of TDD, which whereas aids in creating cleaner and more maintainable code and helps in thinking the solution through before implementation it makes "getting things done" considerably slower especially for the uninitiated, I honestly I would not be surprised that many who complete this course in similar setting as I was have had to ditch TDD and hack the system together just to get it done.

Amount of effort

The total amount of effort on this project stacked up to 65 hours

Notes on docker

Up & Down

```
docker-compose up --build -d
```

remove -d if you want to see logs from the instances

when you want to take the service run in root

```
docker-compose down
```

On volumes

Volumes can be shared between containers, there are unnamed and named containers. Containers can be initialized with data or they can be filled with data afterwards. For instance in this project we are sharing the rabbitmq/index.js file with a volume but just initialing an empty data volume to be shared in runtime.

Volumes are stored on the host machine and can , but normally should not, be accessed as sudo

```
$ sudo su
$ cd /var/lib/docker/volumes
$ ls
```

Useful commands

NOTE! /bin/ash for alpine /bin/bash for other linux distros

Ssh into a running container with `docker exec -it <container name> /bin/bash`

Run an existing image in interactive mode `docker run -it <container name> /bin/ash`

Inpect anything `docker inspect <container name>` `docker inspect <image name>`

List available things running containers `docker ps` all containers `docker container ls --all` all images `docker image ls --all` all volumes `docker volume ls`

Remove non essential stuff remove unused containers `docker container prune` remove unused images `docker image prune` remove unused volumes `docker volume prune` if you want to remove absolutely all of one category add the `--all` flag

Kill all running containers docker container kill $(docker ps -q)

Sources

https://stackoverflow.com/questions/31746182/docker-compose-wait-for-container-x-before-starting-y https://stackoverflow.com/questions/47710767/what-is-the-alternative-to-condition-form-of-depends-on-in-docker-compose-versio https://stackoverflow.com/questions/57065750/docker-compose-volume-overrides-copy-in-multi-stage-build https://docs.docker.com/config/containers/start-containers-automatically/#use-a-restart-policy https://docs.docker.com/engine/reference/commandline/container_prune/

RabbitMQ

RabbitMQ is one of the most popular open source message brokers including message queuing, topic based communication with included support for data safety. It is used by small to large enterprises, it is highly performant, scalable, lightweight and easy to deploy both on premises and to the cloud.

Publisher

Sends messages to an exchange or directly to a queue

Consumer

Initializes a named or unnamed queue which receives either direct messages from a producer or messages matching the subscribed topic from an exchange

Queue

a client can be part of a queue and receive messages from it.

Receiving messages

How we actually receive a message depends on how we have subscribed, but given that this is how we receive and handle a message

```
channel.consume(queue,((message) => {
    if (message !== null) {
      channel.ack(message) // https://www.rabbitmq.com/confirms.html
      messageCallback(null, message.content.toString())
```

```
    }
}))
```

Sending messages

You can send a message to a queue with

```
const options = {
    persistent:true // The message will survive broker restarts provided
it’s in a queue that also survives restarts.
  }
channel.sendToQueue(queue,Buffer.from(message),options)
```

Publishing a message

You can publish a message in an exchange with.

```
channel.publish('exchange-name','', Buffer.from('something to do'))
```

instead of the empty string, you can define a routing key to deliver the message only to queues that are subscribed to the route

Exchange

a queue can be part of an exchange, the exchange can be in different modes.

Fanout

In this mode the exchange sends all messages to all queues

```
// initialize what exchange we are part of
channel.assertExchange('name-of-exchange', 'fanout', {
  durable: false
});

// if you leave the queue name empty, the server generates one on random
// setting exclusive = true scopes the queue to the connection
channel.assertQueue('', { exclusive: true }, (err, q) => {
  // we leave the last value empty so that we will receive all the messages
  channel.bindQueue(q.queue, 'name-of-exchange', '');
})
```

Direct

In this mode we send messages only directly to queues that have bound themselves to listen to the messages:

```
// initialize what exchange we are part of
channel.assertExchange('name-of-exchange', 'direct', {
  durable: false
});

// if you leave the queue name empty, the server generates one on random
channel.assertQueue('', { exclusive: true }, (err, q) => {
  channel.bindQueue(q.queue, 'name-of-exchange', route);
})
```

Topic

Sometimes a simple route is not enough, for instance if we are routing logs to different queues depending on the severity, we might also want to differentiate on the origin. A warning log from a critical system could for instance want to be routed to a queue that normally handles only errors.

Sources

https://www.rabbitmq.com/tutorials/tutorial-one-javascript.html
http://www.squaremobius.net/amqp.node/channel_api.html#channel

Working with multiple git remotes

Adding a remote

```
# git remote add REMOTE-ID REMOTE-URL
# A common practice is to name the primary remote as origin

# Add remote 1: GitHub.
git remote add origin git@github.com:jigarius/toggl2redmine.git
# Add remote 2: BitBucket.
git remote add upstream git@bitbucket.org:jigarius/toggl2redmine.git
```

List all remotes

```
git remote -v
```

Remove a remote

```
# git remote remove REMOTE-ID
git remote remove upstream
```

Change url of an existing remote

```
# The syntax is: git remote set-url REMOTE-ID REMOTE-URL
git remote set-url upstream git@foobar.com:jigarius/toggl2redmine.git
```

Push to single remote

```
# git push REMOTE-ID
git push upstream
```

Push to multiple remotes

```
# Create a new remote called "all" with the URL of the primary repo.
git remote add all git@github.com:jigarius/toggl2redmine.git
# Re-register the remote as a push URL.
git remote set-url --add --push all
git@github.com:jigarius/toggl2redmine.git
# Add a push URL to a remote. This means that "git push" will also push to
this git URL.
git remote set-url --add --push all
git@bitbucket.org:jigarius/toggl2redmine.git
```

TODO

- [ ] And hot code reloading to all scripts
- [ ] Consider using Lerna monorepo with Yarn Workspaces and changing the modules to be published as NPM packages
- [ ] Write unit tests for S3 service
- [ ] Add integration tests