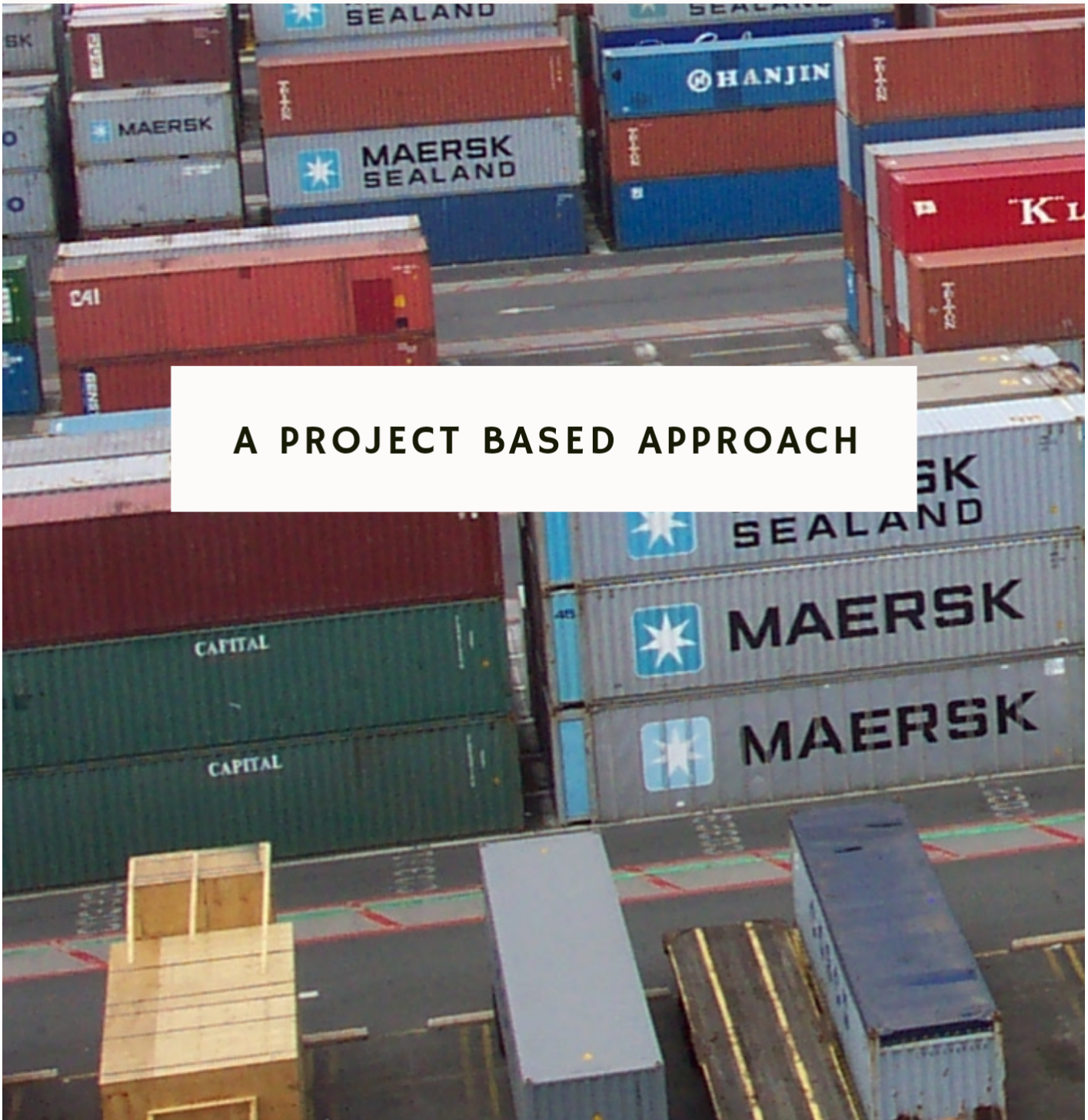
SYSTEMS INTEGRATION



A PROJECT BASED APPROACH

Systems Integration

A Project Based Approach

Ryan Tolboom

Table of Contents

Legal

Preface

The goal of this text is to provide a practical introduction to systems integration by designing and implementing an actual system. In accordance with the show-me-the-code attitude of the open source software movement, all of the code for the text and examples are available at https://github.com/rxt1077/it490. The philosophy of this text can probably best be summed up in a quote:

> Knowledge isn't power until it's applied.
>
> — Dale Carnegie

A computer that meets these minimum specifications will be required to complete the coursework.

Chapter 1. Git

1.1. Version Control

In the simplest sense, version control tracks changes to a group of files. Building off of this premise, teams can use version control to cooperative change a group of files and revert to a previous state if needed.

Some of the benefits of a version control system can be understood if you consider a few examples:

Example 1. Programming Project

> You are working on a project for your CS101 class and you need to write a Python program that plays tic-tac-toe. It must support player-vs-computer *and* player-vs-player. It's due in two days. On the first day you write the initial code and implement player-vs-player. It works great and you fall asleep knowing tomorrow you will finish it and turn it in on time. The next morning you update it to support player-vs-computer and *everything* stops working. What do you do now?

If you were working with a version control system, you could easily roll-back your changes and see what went wrong.

Example 2. Working with a Team

> You are working working with a team of people to implement a complex system that utilizes several files in several different directories. How do you make sure everyone has the most up-to-date version of the files? What happens if two people work on the same file at the same time?

In a version control system, you could set up a centralized repository for the files and have everyone pull from one location. In the case of two people working on the same thing at the same time, a version control system could handle merging their changes.

1.2. Getting Git

Depending on the OS you are using, there are a few different ways to install git on your machine:

Windows

- git for windows: Installs git, git BASH, and a GUI. The git command can then be run from PowerShell, CMD, or the BASH shell (which it installs).

Mac

- git for Mac Installer: Provides an easy installer for git on MacOS.
- Xcode: Xcode installs a command line git and you may have it installed already.

Linux

- Basically all distributions have git available in their standard package manager. Chances are, if you're running Linux you have it installed already, so I'll take the opportunity to highlight the strangest distro I can think of: You can install git in Hannah Montana Linux with the command `apt-get install git`.

1.3. Basic Git Actions

1.3.1. Creating a Repository

Any directory can be made into a git repository by running the `git init` command. This will add the `.git` directory which stores information about the state of the repository and certain user variables.

1.3.2. Cloning a Repository

If you want to make a copy of an already existing repository, typically done when you *start* working on a project, the `git clone` command will make a copy of a repository using any supported protocol.

1.3.3. Tracking / Staging Files

You need to tell git which files you want to keep track of and when you want to stage them to be committed. These both use the same command `git add`. The first time you call `git add` it begins tracking the file and stages it for a commit. The second time you call `git add` it tells git that you want to commit any changes to that file.

1.3.4. Committing Changes to a Repository

Once you have made some changes to your repository and staged those files with `git add` you can use the `git commit` command to *commit* your changes. All commits must have a message, and git will use a default editor depending on your installation. This can be changed. To complete the commit, add a message, save the file, and exit the editor.

1.3.5. Setting Up a Remote

Git repositories are often linked to a remote repository. This could be a service, like GitHub, GitLab, or SourceHut or more simply another server that the team has access to. The `git remote add origin` command adds a remote URL that is the default target for actions. You can then `git push` your changes to the remote or `git pull` to get the latest changes from the remote. The `git clone`

command automatically sets the origin.

1.4. Example

Let's take a look at an example of two people, Jessica and Darsh, working with the same remote repository:

Jessica's First Session (PowerShell)

```
PS jess> mkdir example ①


    Directory: jess


Mode                 LastWriteTime         Length Name
----                 -------------         ------ ----
d-----        4/22/2020  10:02 PM                example


PS jess> cd example
PS jess\example> git init ②
Initialized empty Git repository in jess/example/.git/
PS jess\example> Set-Content -Path 'test.txt' -Value 'Hello from git!' ③
PS jess\example> git add . ④
PS jess\example> git commit -m "Initial Commit" ⑤
[master (root-commit) 46c7c75] Initial Commit
 1 file changed, 0 insertions(+), 0 deletions(-)
 create mode 100644 test.txt
PS jess\example> git remote add origin ssh://git@192.168.10.1/home/git/example.git ⑥
PS jess\example> git push origin master ⑦
git@192.168.10.1's password:
Enumerating objects: 3, done.
Counting objects: 100% (3/3), done.
Writing objects: 100% (3/3), 249 bytes | 249.00 KiB/s, done.
Total 3 (delta 0), reused 0 (delta 0)
To ssh://192.168.10.1/home/git/example.git
 * [new branch]      master -> master
```

① Jessica will be creating the repository, so she makes a new directory

② Inside the directory, she uses `git init` to initialize it

③ She adds some content so she has something to commit

④ The form `git add .` means *stage all files in this directory*. It is the most common invokation of `git add`.

⑤ Jessica commits her work. The `-m` option allows her to add a commit message without needing to open an editor.

⑥ She adds a remote as the default. This *does* require configuration on the remote server, a local machine in our case, but we will talk about how that is usually handled in the GitHub section.

⑦ Finally she pushes her changes to the remote so that Darsh can get them.

Darsh's Session (BASH)

```
darsh@laptop:~$ git clone ssh://git@192.168.10.1:/home/git/example.git ①
Cloning into 'example'...
remote: Enumerating objects: 3, done.
remote: Counting objects: 100% (3/3), done.
remote: Total 3 (delta 0), reused 0 (delta 0)
Receiving objects: 100% (3/3), done.
darsh@laptop:~$ cd example ②
darsh@laptop:~/example$ cat test.txt
Hello from git! ③
darsh@laptop:~/example$ echo "Hello Jess!" >> test.txt ④
darsh@laptop:~/example$ git add . ⑤
darsh@laptop:~/example$ git commit -m "Added my message"
[master 55dc946] Added my message
 1 file changed, 1 insertion(+)
darsh@laptop:~/example$ git push
Counting objects: 3, done.
Writing objects: 100% (3/3), 271 bytes | 271.00 KiB/s, done.
Total 3 (delta 0), reused 0 (delta 0)
To ssh://192.168.10.1:/home/git/example.git
   182a481..55dc946  master -> master
```

① Darsh isn't creating a new repository so he uses the `git clone` command to clone the repository Jessica has made.

② By default, cloned repositories are put in their own directory based on the repository name. You can specify a different directory by adding an argument after the URL: `git clone ssh://git@192.168.10.1:/home/git/example.git new-directory`

③ Jess's content is there!

④ Darsh appends a message of his own.

⑤ Then he follows the standard add, commit, push work flow to sync his changes.

Jessica's Second Session (PowerShell)

```
PS jess\example> Get-Content -Path 'test.txt'
Hello from git! ①
PS jess\example> git pull origin master ②
git@192.168.10.1's password:
From ssh://192.168.10.1/home/git/example
 * branch            master     -> FETCH_HEAD
Updating 182a481..55dc946
Fast-forward
 test.txt | 1 +
 1 file changed, 1 insertion(+)
PS jess\example> Get-Content -Path 'test.txt'
Hello from git! ③
Hello Jess!
```

① When Jess goes to check on Darsh's work, it isn't there! Why?

② Because she hasn't pulled from the remote yet.

③ Once she does, she can see Darsh's addition

This scenario begs the question, "What would happen if Jess didn't pull Darsh's work and kept working on her local, unsynced copy?" Assuming they were both working on the same file, when Jess goes to push there would be a merge conflict. Git is very good at resolving conflicts and team members tend to be working on different parts of the codebase, making the resolution simpler.

1.5. Resources

- The entire Pro Git Book can be found online. It is a comprehensive text that will cover much more than the brief outline presented here.
- GitHub has some excellent and interactive resources for learning to use git.

1.6. Questions

1. *What are the advantages of using version control?*

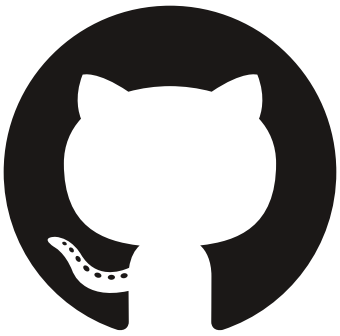
2. *What does it mean that files are staged for a commit?*

3. *What are the two things that the `git add` command can do?*

4. *How do you create a new repository in a directory?*

5. *What is a remote and what does the `git push` command do?*

Chapter 2. GitHub



2.1. Purpose

GitHub is a service that provides a space for remote git repositories. It features an extensive web interface and several project management features.

GitHub has become a popular for open-source projects and is a great way to showcase your projects to prospective employers. It is free to sign up for GitHub and we will be using it for project management in this course. I suggest you sign up with your .edu email address and UCID as your username to keep things simple.

2.2. Remote Repositories

To set up a remote repository in GitHub, follow these steps:

1. Create a local git repository as shown in the previous section. Choose a repository name that is simple. Avoid spaces or trailing dashes as Docker Compose may have troulbe with them.
2. Sign in to GitHub and navigate to Create a New Repository
3. Put in the your repository name (make sure it matches your local repository name) and a description.
4. GitHub now offers unlimited private repositories with unlimited collaborators. This means you could complete your project in a private repository, just be sure to add your instructor as a collaborator. Later when you want to showcase your work, it may be a good idea to make this repository public.
5. You can also work in a public repository and it can be good practice for learning to separate your secrets from your commits. You will still need to add group members as collaborators, but your instructor should be able to have ready-only access without any additional setup.
6. Once you click "Create Repository", instructions will be provided for setting the remote on your local repository. It is very similar to the scenario covered in the previous section. Follow the directions.
7. Now your group members should be able to `clone` the repository, but they will not be able to make commits until you invite them as collaborators.
8. Go to *Settings (top right gear icon)* → *Manage access* → *Invite a Collaborator* within the GitHub web interface for your repository. Add all of your group members as collaborators.

2.3. Issues

GitHub has a built-in bug tracker called *Issues*. You can find it next to the *Code* tab, under the repository name when viewing a repository. An issue is typically a bug that needs to be fixed or a feature that needs to be implemented. It can be assigned to a group member who can then close it when it is completed. It can also be linked to a milestone, which can be thought of as as group of things that need to be done to reach the next phase.

We will be using the GitHub Issues to monitor individual contributions to a project and to see how well a team functions. Do not be afraid to create issues and use the discussion features inside of them. They help document your progress. You will also have milestones assigned as you progress through the text. You should create those milestones in GitHub and assign the goals accordingly.

2.4. Pull requests

For complex projects or projects that have external contributers GitHub supports a fork-based pull request (PR) workflow. Although we probably won't be using it too much, it is helpful to know how it works in case you end up working on larger projects, you want to contribute to another project, or your instructor wants to contribute to your project.

In GitHub, a typical PR workflow looks like this:

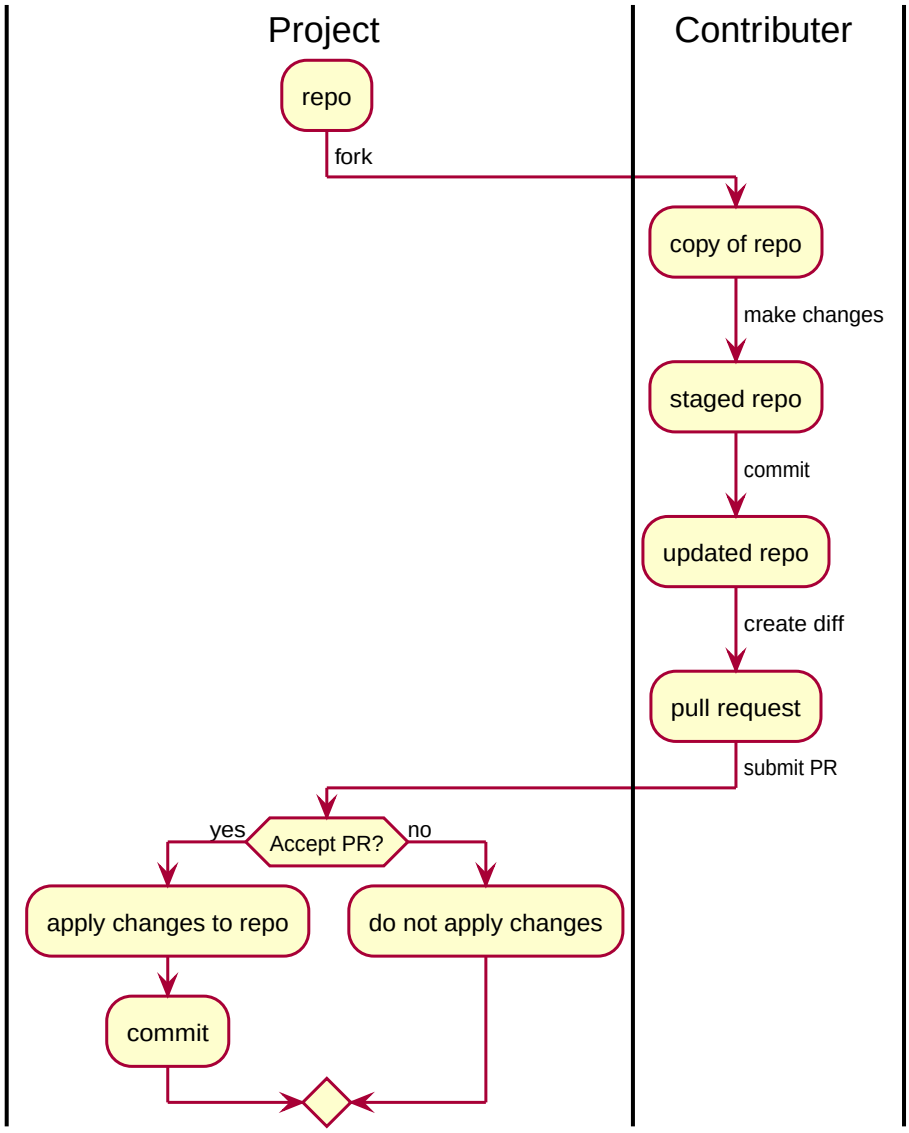


Figure 1. Pull Request

A contributer forks a project (makes their own personal copy), which is as easy as clicking the *Fork* button in the upper-right when viewing a project repository. They change the parts of the repository that they want to in their personal copy and commit their changes. Then they click on *Pull Requests → Create Pull Request* on the project's repository. GitHub defaults to creating PRs across branches (a good technique when working on a large project with lots of contributers), but you can select PRs across forks as well. The owner of the project can review the PR and if they like the changes they have the option to merge them with their repository.

2.5. Documentation

GitHub supports several styles of documentation, but the most common is Markdown. Files written in markdown and ending in the `.md` extension will be rendered and displayed when viewed in the GitHub web interface. If a file named `README.md` exists in a directory, it will be automatically displayed at the bottom of a directory listing. This makes it easy to build documentation right into your repository. Learn markdown and be sure to have a `README.md` in your repository.[1]

2.6. Resources

- Mastering Issues
- Making a Pull Request
- About Pull Requests
- The Markdown Guide

2.7. Questions

1. *What does GitHub provide for a project?*

2. *What is the difference between using git and GitHub?*

3. *A new member joins your team. As the maintainer of the repository on GitHub, what steps do you need to take so that they have commit access to the repository? What steps does the group member need to take to get set up?*

4. *What is the purpose of issues in GitHub?*

5. *Why might a team want to use pull requests instead of adding all contributers as collaborators to a project?*

[1] If you're looking to take things a bit further asciidoc, reStructuredText, and scribble are worth exploring too. This book is written in asciidoc.

Chapter 3. YAML

3.1. Introduction

In the long-standing tradition of informal, recursive acronyms, YAML stands for YAML Ain't Markup Language. It is designed to be a plain text way to represent complex objects. It is easier to read than JSON, but not as complex as XML. YAML uses indentation to specify scope, like Python, and therefore *spacing matters*.

The vast majority of what we will be creating is written in YAML so it pays to give it at least a cursory treatment. It's even become a bit of an inside joke that modern system architects are simply YAML engineers.

3.2. Parts of a YAML Stream

This section is parts of a YAML *stream*, not a YAML *document* because technically a single YAML file could have multiple Documents.

Let's look at some sample streams to get a clearer picture of how YAML is used:

Sample Docker Compose YAML

```
# source: https://docs.docker.com/compose/gettingstarted/ ①
--- ②
version: '3' ③
services: ④
  web:
    build: .
    ports:
    - "5000:5000" ⑤
  redis:
    image: "redis:alpine"
```

① Anything following a `#` in YAML is considered a comment. Don't be afraid to use them!

② YAML documents start with `---` and optionally end with `...`. This allows multiple documents to included in a stream.

③ `keyword: value` signals a mapping. This mapping maps the keyword `version` to the string `'3'`.

④ Mappings can be nested. This mapping maps the keyword `services` to mappings with keywords

`web` and `redis`.

⑤ Sequences can be shown as a block of lines starting with `-`. In this case `ports` is mapping to a sequence with one item, the string `"5000:5000"`.

Sample Kubernetes YAML

```
# source: https://kubernetes.io/docs/tasks/configure-pod-container/configure-pod-
configmap/
apiVersion: v1
kind: ConfigMap
metadata:
  creationTimestamp: 2016-02-18T18:52:05Z ①
  name: game-config
  namespace: default
  resourceVersion: "516"
  uid: b4952dc3-d670-11e5-8cd0-68f728db1985
data:
  game.properties: | ②
    enemies=aliens
    lives=3
    enemies.cheat=true
    enemies.cheat.level=noGoodRotten
    secret.code.passphrase=UUDDLRLRBABAS
    secret.code.allowed=true
    secret.code.lives=30
  ui.properties: |
    color.good=purple
    color.bad=yellow
    allow.textmode=true
    how.nice.to.look=fairlyNice
```

① YAML supports timestamp types.

② The `|` indicates block scalar style. The keyword `game.properties` is mapped to a string where the newlines are preserved but the leading spaces are removed. The string, with newlines shown as '\n', is 'enemies=aliens\nlives=3\nenemies.cheat=true\n...'. This is a common way of defining files or scripts within YAML.

3.3. Editors

Given YAML's strict whitespace requirements, you will need to use a text editor that supports configuring spacing and expanding tabs into spaces. At a minimum, you should be able to do the following with your text editor:

- Translate tab keystrokes into spaces. This is sometimes referred to as *expanding tabs*. YAML files that mix tabs and spaces will not work.
- Adjust tab spacing. Typically you will see YAML files with two spaces of indentation for their blocks. This allows you to have many nested blocks and not have to worry about lines wrapping or scrolling to the right.

- Increase / Decrease the indentation level of several lines at once. As you make changes, you may have to change the indentation level of a block being able to do this quickly, without having to visit every line will save you time.
- Cut/Copy/Paste - You should be able to copy things between your web browser and the file your are editing. If you can copy things between multiple tabs/buffers in your text editor, even better.
- Convert between DOS/UNIX line endings. Most of the tools you will be working with come from the UNIX world, where a line ends with '\n'. Some older DOS utilities still end lines with '\r\n'. You need to be able to save documents with UNIX line endings in your text editor.

There are many editors that meet these requirements. Choosing an editor is a matter of personal taste and the subject of unending flame wars. With this in mind the following list is not meant to be exhaustive and I'm sure the comments may be subject of some debate. Popular editor choices:

- **vim**/**neovim**/ **vi** - Some form of vi is almost always installed on any *NIX/BSD system. Knowing how to use it can be a major lifesaver. You can also find versions for Windows. Since most of your work will be in a terminal, having an editor that runs directly inside a terminal can be an advantage. All that being said, the learning curve is steep. If you are interested in learning vi, I'd suggest either vimtutor or `:Tutor` inside neovim.
- **Visual Studio Code** - vscode is more akin to a modern IDE. It is rapidly gaining more adoption and is certainly worth checking out if that is the type of experience you are looking for.
- **Notepad++** - Notepad++ is a popular Windows GUI text editor. It starts quickly, and many things work right out of the box. If you want something like notepad, but a little more versatile (the next iteration you could say... get it?), then this is for you.
- **TextMate** - TextMate is a popular MacOS GUI text editor. It is simple to get started, but offers the advanced features you may need as you progress.

3.4. Resources

- The Official YAML Web Site
- YAML Multiline Strings

3.5. Questions

1. *How does YAML signify different blocks?*

2. *Are nested structures possible in YAML?*

3. *What are the two components of a YAML mapping?*

4. *How would you comment out a line in a YAML file?*

5. *What does the expandtab or "replace by spaces" option do in a text editor and why is it important for YAML?*

Chapter 4. Midterm Check In

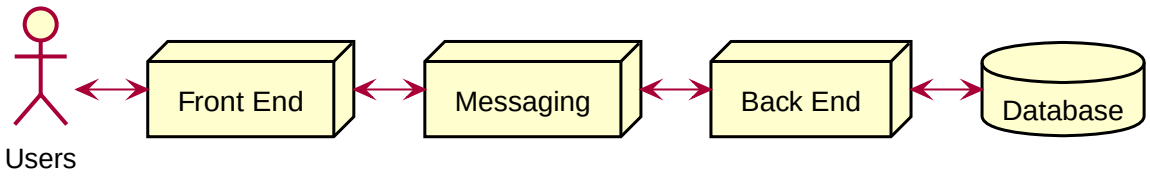
This section summarizes the content of the class so far and provides an example of a system that meets the midterm goals. This example is just one way to solve the problem. It's not the only way, in fact it's not even the best way. The example's purpose is to show you a solution that avoids common pitfalls. Hopefully you can integrate some of the lessons of this example into your project.

4.1. Overview

At this point you should have a working system that can be pulled from a git repository and brought up with `docker-compose up`. As a reminder, the midterm deliverables used to the project are:

- A **Front End** that interacts with the user via HTTP and communicates with the messaging container.
- **Messaging** (RabbitMQ) that brokers the exchange of information between the front end and the back end.
- **Database** (PostgreSQL, MariaDB, MySQL) that only **Back End** uses for storage of persistent information. All stateful information should be stored in a volume.
- A **Back End** that gathers information from your data sources, reads and writes from the database, and interacts with the messaging container.
- These four services work with each other to provide a web-based registration and authentication system.

The project is structured in such a way that **Front End** and **Back End** never communicate directly and **Back End** is the only service that can write to the **Database**:



This allows for more scalability when we introduce replication in the second half of the semester.

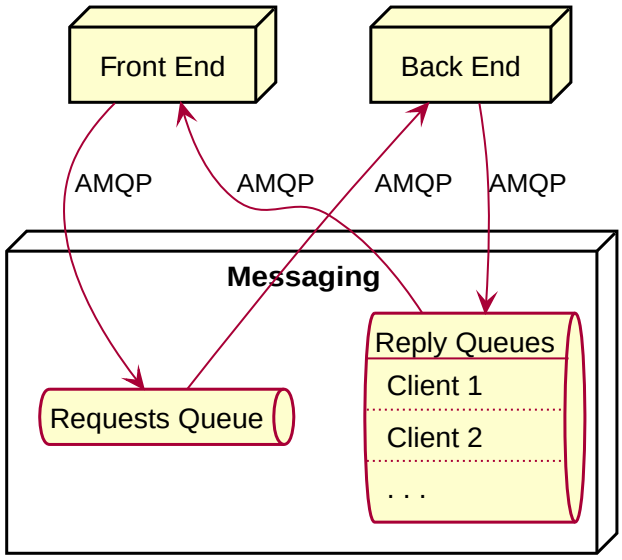
An example directory structure is shown below:

```
|    .env
|    docker-compose.yml
|
├──────back_end
|        app.py
|        Dockerfile
|        requirements.txt
|
├──────db
|        Dockerfile
|        setup.sql
|
└──────front_end
     |    app.py
     |    Dockerfile
     |    requirements.txt
     |    wait-for-it.sh
     |
     └──────templates
            base.html
            login.html
            register.html
```

Notice the `.env` file in the directory structure. docker-compose will load environment variables from this file that you can then use in the `docker-compose.yml` file. This keeps you from having to repeat yourself when multiple services need the same information. For example, both **Database** and **Back End** need to know the `POSTGRES_PASSWORD`. It also allows you to have a single secret file that you can put in `.gitignore` to keep out of your repository.

4.2. Messaging

In our example, the **Messaging** can be run straight from the RabbitMQ Docker Hub image via the `docker-compose.yml` file, hence the absence of a `messaging` directory with a `Dockerfile` in the directory structure. The image allows for sufficient configuration via its environment variables. At this point it is recommended that you still run the management interface and forward the management interface port, 15672, so that you can see how the queues are being used. The messaging service will be used in the request / reply pattern detailed in the diagram below:



Fortunately this works out-of-the-box since queue creation is handled by the by the clients. The service simply has to be up and running to function.

The service definition in `docker-compose.yml` can be seen here:

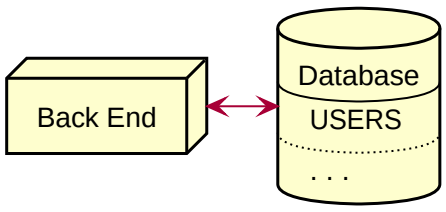
docker-compose.yml (excerpted)

```
    messaging:
        image: 'rabbitmq:3-management'
        environment:
            RABBITMQ_DEFAULT_USER: ${RABBITMQ_DEFAULT_USER}
            RABBITMQ_DEFAULT_PASS: ${RABBITMQ_DEFAULT_PASS}
            RABBITMQ_ERLANG_COOKIE: ${RABBITMQ_ERLANG_COOKIE}
        ports:
            - 15672:15672
```

Moving forward, we will migrate from a single RabbitMQ instance to a cluster. Familiarize yourself with the RabbitMQ Clustering Guide and then see if you can create a three node cluster with docker-compose.[2] You may still be able to do this with just environment variables in the `docker-compose.yml` file. If you need to, feel free to create new container images in separate directories with their own Dockerfiles. There are also plenty of good web resources to explore.

4.3. Database

Database is responsible for storing the persistent information the system uses. It only communicates with **Back End**.



In this example, **Database** creates a database and the appropriate tables *if* the database is currently empty. This can be done by placing the SQL file that we want executed in `/docker-entrypoint-initdb.d/` of the image. Our `db/Dockerfile` handles copying our initialization SQL appropriately:

db/Dockerfile

```
FROM postgres
COPY setup.sql /docker-entrypoint-initdb.d/
```

db/setup.sql

```
CREATE DATABASE example;
\c example
CREATE TABLE users(
    email VARCHAR(255) PRIMARY KEY,
    hash VARCHAR(255) NOT NULL
);
```

The schema in this example is quite simple, consisting of a database and a single table for holding emails and hashes. It should be noted that the `\c` command is specific to PostgreSQL and it is the equivalent of a `USE` statement in MySQL, meaning use that particular database.

The relevant service definition part of the `docker-compose.yml` file can be seen here:

docker-compose.yml (excerpted)

```
    db:
        build: ./db
        environment:
            POSTGRES_PASSWORD: ${POSTGRES_PASSWORD}
        volumes:
            - data-volume:/var/lib/postgresql/data
```

The database files are stored in a Docker volume named `data-volume` and the password for our database is loaded from an environment variable defined in `.env`.

The adminer image is a great way to see what's going on in your database. It provides a web interface to many different types of databases that can be easily accessed via port 8080. See the example `docker-compose.yml` file for an example of its use.

Moving forward, we should begin to explore how we can implement replication to make **Database** more scalable and resilient. Try to get a master / slave configuration set up in a `docker-compose.yml`. How does replication help our system? Will we need any other pieces to maximize its potential? Once again, there are many good web resources on the topic.

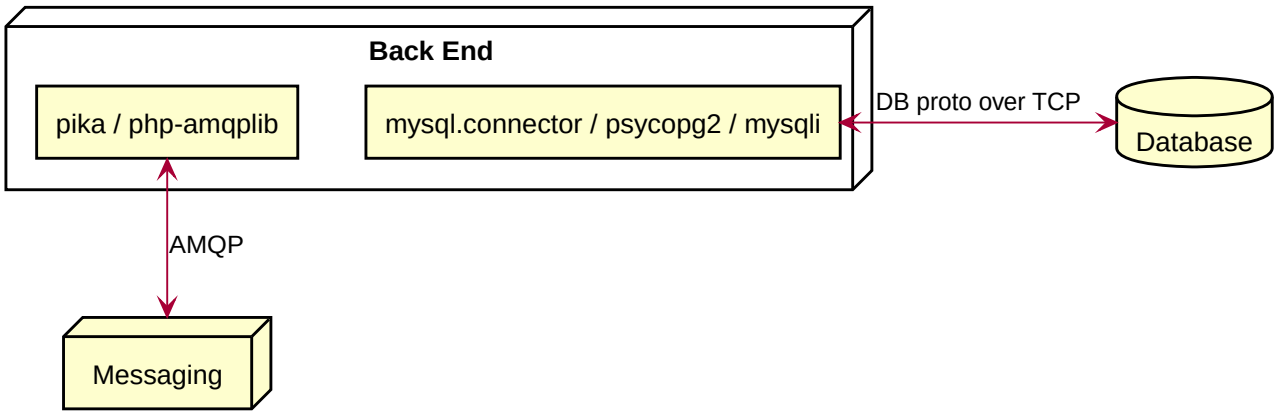
4.4. Back End

Back End brokers the exchange of information between **Messaging** and **Database**. It also provides an area to perform the tasks needed to support the complete system such as web scraping or computation. At this point, our example does not include any of the latter and mainly focuses on storing / utilizing authentication information in the database.

You can think of **Back End** as providing an API that is accessible through **Messaging**. Any language can be a good choice for the **Back End** as long as it has libraries to interface with **Database** and **Messaging**.

Popular Libraries used by **Back End**

- Python
  - psycopg2 (PostgreSQL)
  - mysql.connector (MySQL / MariaDB)
  - pika (RabbitMQ)
- PHP
  - mysqli (MySQL / MariaDB)
  - php-amqplib (RabbitMQ)



The code for the **Back End** example is entirely contained in `back_end/app.py`. **Back End** starts by connecting to both **Messaging** and **Database** using the pika and psycopg2 libraries respectively. With Docker Compose you don't know when the services will become available so the example repeatedly attempts to connect, waiting up to 60 seconds and backing off exponentially each time. Below is an example of typical startup output:

```
> docker-compose logs --tail=100 back_end | Select-String -Pattern root

back_end_1   | INFO:root:Waiting 1s...
back_end_1   | INFO:root:Connecting to the database...
back_end_1   | INFO:root:Connecting to messaging service...
back_end_1   | INFO:root:Waiting 2s...
back_end_1   | INFO:root:Connecting to the database...
back_end_1   | INFO:root:Connecting to messaging service...
back_end_1   | INFO:root:Waiting 4s...
back_end_1   | INFO:root:Connecting to the database...
back_end_1   | INFO:root:Connecting to messaging service...
back_end_1   | INFO:root:Waiting 8s...
back_end_1   | INFO:root:Connecting to the database...
back_end_1   | INFO:root:Connecting to messaging service...
back_end_1   | INFO:root:Starting consumption...
```

The example then creates the required database cursor, messaging channel, messaging queues, and sets up a callback for messages arriving in the `requests` queue. This is where the majority of work is performed and the function can be seen here:

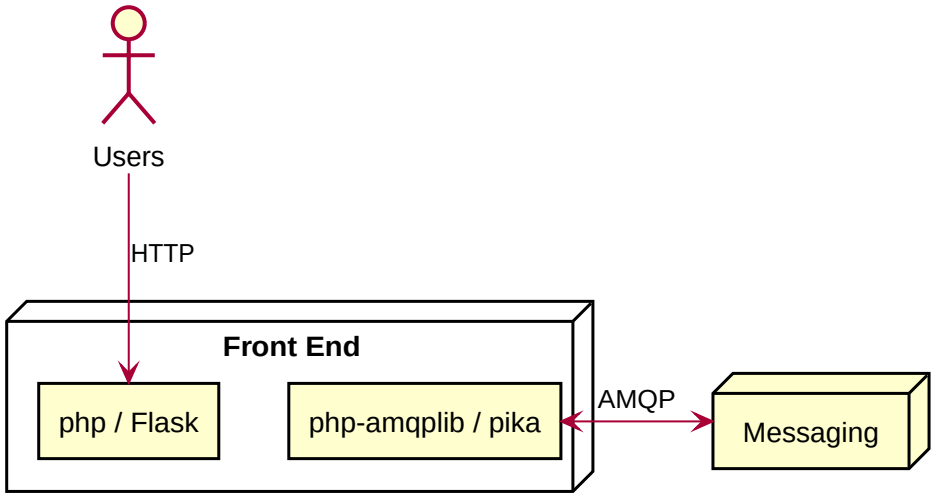
back_end/app.py (excerpted)

```
def process_request(ch, method, properties, body):
    """
    Gets a request from the queue, acts on it, and returns a response to the
    reply-to queue
    """
    request = json.loads(body)
    if 'action' not in request:
        response = {
            'success': False,
            'message': "Request does not have action"
        }
    else:
        action = request['action']
        if action == 'GETHASH':
            data = request['data']
            email = data['email']
            logging.info(f"GETHASH request for {email} received")
            curr.execute('SELECT hash FROM users WHERE email=%s;', (email,))
            row =  curr.fetchone()
            if row == None:
                response = {'success': False}
            else:
                response = {'success': True, 'hash': row[0]}
        elif action == 'REGISTER':
            data = request['data']
            email = data['email']
            hashed = data['hash']
            logging.info(f"REGISTER request for {email} received")
            curr.execute('SELECT * FROM users WHERE email=%s;', (email,))
            if curr.fetchone() != None:
                response = {'success': False, 'message': 'User already exists'}
            else:
                curr.execute('INSERT INTO users VALUES (%s, %s);', (email, hashed))
                conn.commit()
                response = {'success': True}
        else:
            response = {'success': False, 'message': "Unknown action"}
    logging.info(response)
    ch.basic_publish(
        exchange='',
        routing_key=properties.reply_to,
        body=json.dumps(response)
    )
```

Notice that psycopg2 functions are used to put variables into the SQL statements. Do **NOT** use Python string formating to build your SQL statements. You may be thinking that we are in **Back End** and the parameters we receive are already sanitized by **Front End** but this is not always the case.

4.5. Front End

The **Front End** can be created using any web framework, but the most popular choices are PHP or Flask. The most popular Docker Hub images for those frameworks are php:apache and python respectively. For interacting with **Messaging** there are a few options, but groups tend to gravitate towards php-amqplib for PHP and pika for Python Flask. This is probably due to the fact that the documentation for RabbitMQ references those libraries.



A custom `front_end/Dockerfile` is used for creating the image:

front_end/Dockerfile

```
FROM python
COPY . /app
WORKDIR /app
RUN pip install -r requirements.txt
ENV FLASK_APP=app.py
CMD ["./wait-for-it.sh", "messaging:5672", "--", \
     "flask", "run", "--host=0.0.0.0"]
```

A script called `wait-for-it.sh` is included with the image. It is used as recommended by the Docker documentation to make sure the **Messaging** is up before **Front End** starts. This way **Front End** will give errors to users who attempt to use it before the full system has been brought up.

`front_end/Dockerfile` is built in the service section of the `docker-compose.yml` file:

docker-compose.yml (excerpted)

```
    front_end:
        build: ./front_end
        ports:
            - 5000:5000
        environment:
            RABBITMQ_DEFAULT_USER: ${RABBITMQ_DEFAULT_USER}
            RABBITMQ_DEFAULT_PASS: ${RABBITMQ_DEFAULT_PASS}
            FLASK_ENV: development
            FLASK_SECRET_KEY: ${FLASK_SECRET_KEY}
        volumes:
            - "./front_end:/app"
```

There are a few things in the `docker-compose.yml` service definition that should be noted:

- Port `5000` needs to be forwarded as it will be accessed externally
- The `FLASK_ENV` environment variable is useful for development. It causes Flask to print more friendly error messages right inside the web browser.
- `front_end/` is bind mounted to `/app` in the container even though the `Dockerfile` copies those files to the `/app` directory when the image is created. This allows for easier development, the container can be running while you edit the files Flask is using. The development server will automatically restart if changes are detected.

To make communication with **Messaging** a easier, `front_end/messaging.py` defines a `Messaging` class that initializes the connection, shuts down the connection, and provides a send and receive function. All messages are sent to the general `requests` queue and replies are returned via an exclusive reply queue.

front_end/messaging.py

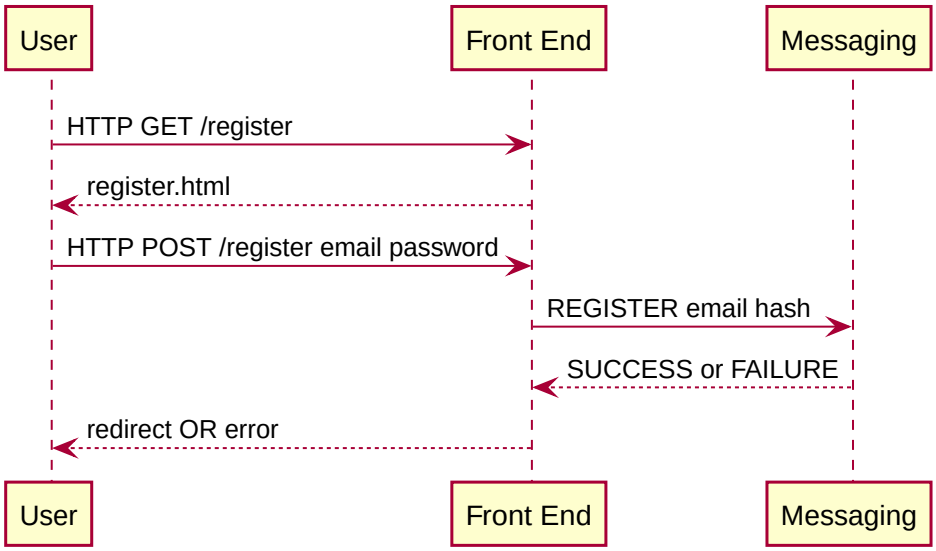
```
import pika
import json
import time
import logging
import os

class Messaging:
    """
    Helper class for dealing with the messaging service
    """
    request_queue_name = 'request'

    # Get credentials from the environment
    credentials = pika.PlainCredentials(os.environ['RABBITMQ_DEFAULT_USER'],
                                        os.environ['RABBITMQ_DEFAULT_PASS'])

    # docker-compose will resolve this host to our messaging service
    host = 'messaging'
```

```
    def __init__(self):
        """
        Establishes connection and creates queues as needed
        """
        logging.info("Messaging: Establishing connection")
        self.connection = pika.BlockingConnection(
            pika.ConnectionParameters(host=self.host, credentials=self.credentials))
        self.channel = self.connection.channel()
        logging.info("Messaging: Creating queues")
        self.channel.queue_declare(queue=self.request_queue_name)
        self.result_queue = self.channel.queue_declare(queue='',
exclusive=True).method.queue

    def __del__(self):
        """
        Closes down the connection
        """
        logging.info("Messaging: Closing down connection")
        self.connection.close()

    def send(self, action, data):
        """
        Sends an action and data to the request queue in JSON. Sets the
        reply_to property to the custom result queue.
        """
        logging.info(f"Messaging: send(action={action}, data={data})")

        self.channel.basic_publish(
            exchange='',
            routing_key=self.request_queue_name,
            properties=pika.BasicProperties(
                reply_to=self.result_queue),
                body=json.dumps({'action': action, 'data': data}
            )
        )

    def receive(self):
        """
        Waits for a single message and returns it. Waits up to 1s, checking
        every 0.1s.
        """
        attempts = 0
        while True:
            method_frame, properties, body = self.channel.basic_get(
                self.result_queue, auto_ack=True)
            if method_frame:
                received = json.loads(body)
                logging.info(f"Messaging: received={received}")
                return received
            elif attempts > 10:
```

```
                logging.info("Messaging: receive did not get message")
                return None
            else:
                time.sleep(0.1)
                attempts += 1
```

Every HTTP request received by **Front End** will result in a full connection sequence with **Messaging** which is not optimal. A better solution is to have the `Messaging` class run in its own thread and maintain a permanent connection.

Let's take a look at a registration sequence involving the **User**, **Front End**, and **Back End**:



The relevant code in `app.py` follows:

front_end/app.py (excerpted)

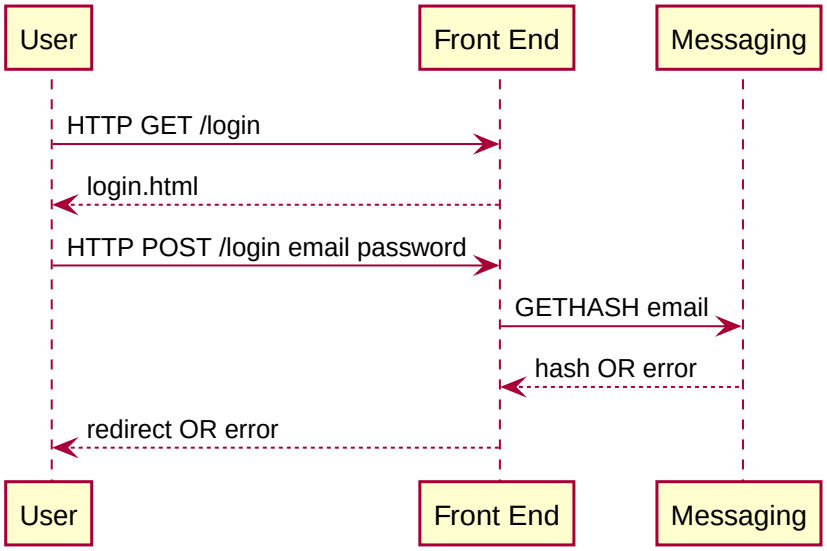
```
@app.route('/register', methods=['GET', 'POST'])
def register():
    if request.method == 'POST':
        email = request.form['email']
        password = request.form['password']
        msg = messaging.Messaging()
        msg.send(
            'REGISTER',
            {
                'email': email,
                'hash': generate_password_hash(password)
            }
        )
        response = msg.receive()
        if response['success']:
            session['email'] = email
            return redirect('/')
        else:
            return f"{response['message']}"
    return render_template('register.html')
```

Fortunately password hashing functions are readily available in the `werkzeug.security` module. Werkzeug is a WSGI utility library that Flask already uses, so we don't need to add anything to `requirements.txt`. We can use the functions `check_password_hash` and `generate_password_hash`.

Do **NOT** store user passwords in cleartext. There are plenty of good hashing options in both PHP and Python. Try to minimize systems that come in contact with unencrypted passwords as well. In this example it is hashed before it is even passed to the **Back End**. For the same reason, in production your users should only be able to connect via TLS (HTTPS). This will secure the channel between **User** and **Front End** which passes the password when the user registers or logs in.

The `register` function will also set `email` in the user session upon successful completion. This is akin to having a user log in automatically once they have created an account. Flask sessions are a good way of storing things on the client that can't be modified. They default to a lifetime of 31 days.

A similar sequence is used to log in a user:



The relevant code in `front_end/app.py` follows:

front_end/app.py (excerpted)

```
@app.route('/login', methods=['GET', 'POST'])
def login():
    if request.method == 'POST':
        email = request.form['email']
        password = request.form['password']
        msg = messaging.Messaging()
        msg.send('GETHASH', { 'email': email })
        response = msg.receive()
        if response['success'] != True:
            return "Login failed."
        if check_password_hash(response['hash'], password):
            session['email'] = email
            return redirect('/')
        else:
            return "Login failed."
    return render_template('login.html')
```

Compared to the `register` function, handling a login is simpler. A few other routes of interest with brief descriptions are:

- `/` - serves the `index.html`
- `/logout` - removes `email` from the user's session and redirects to `/`
- `/secret` - protected by the `@login_required` decorator (see below), serves `secret.html`

Python decorators are used for routing in Flask so it is a natural fit to use them to protect routes as well. The `@login_required` decorator does exactly that:

front_end/app.py (excerpted)

```
def login_required(f):
    """
    Decorator that returns a redirect if session['email'] is not set
    """
    @wraps(f)
    def decorated_function(*args, **kwargs):
        if 'email' not in session:
            return redirect('/login')
        return f(*args, **kwargs)
    return decorated_function
```

Moving forward, we should start thinking about how we are going to transition our development server to a production environment. The built-in Flask server is great for testing our code but we will want to shift to something meant for large scale use as we begin to scale up our application. Some packages to research are NGINX, uWSGI, gunicorn, Apache, and mod_wsgi.

[2] Eventually we will want things to be more scalable than just a static, three node cluster.

Chapter 5. Replication

Replicating a service can provide many benefits. In this section we will replicate a PostgreSQL database service to analyze the advantages and complexity costs. PostgreSQL was purposefully chosen because it leaves much of work, which is understandably outside the purview of a Database Management System (DBMS), to the user. We will be using Docker compose so that we do not have to introduce a new tool as well.

5.1. Background

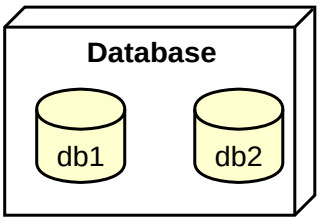


Figure 2. Basic Replication

In the simplest sense, replication is running more than one service for a given component of our system. We can do this in Docker Compose by declaring more than one service:

replication-demo/docker-compose.yml (excerpted)

```
    db1:
        build: ./db
        environment:
            POSTGRES_PASSWORD: asdffdsa
            POSTGRES_REPLICA_PASSWORD: asdffdsa
            POSTGRES_NODES: "db1 db2"
    db2:
        build: ./db
        environment:
            POSTGRES_PASSWORD: asdffdsa
            POSTGRES_REPLICA_PASSWORD: asdffdsa
            POSTGRES_NODES: "db1 db2"
```

A better solution would be to use a more complete orchestration solution than Docker Compose. For syntax that you are used to, see the deploy option and Docker Swarm. You could also bite the bullet and migrate to kubernetes.

In most DBMS, replication can be implemented via Volume Sharing or Hot / Warm Standby:

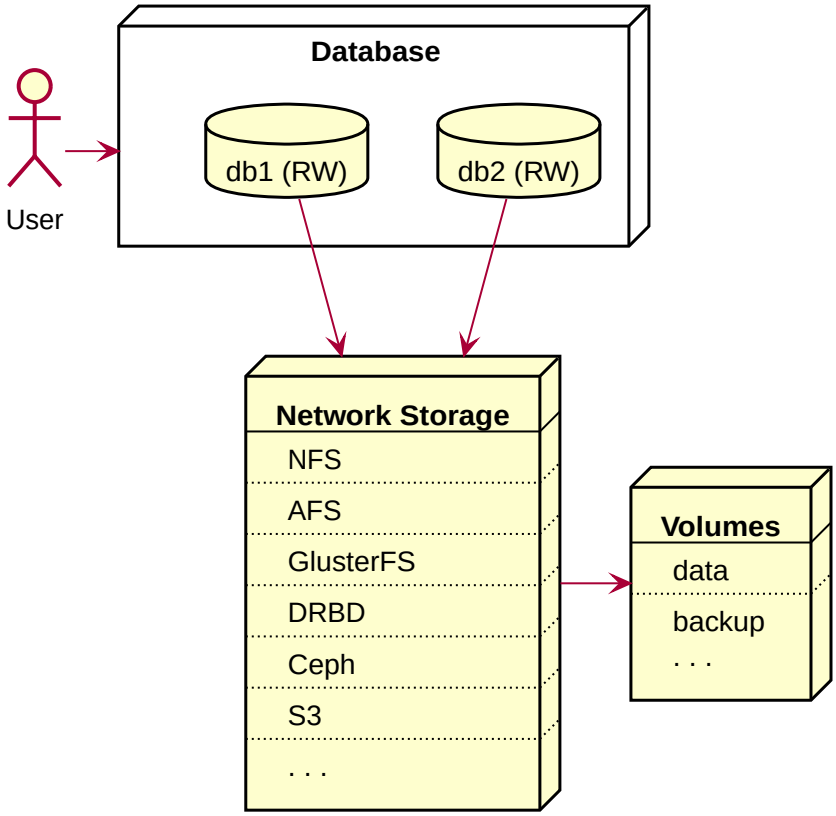


Figure 3. Volume Sharing

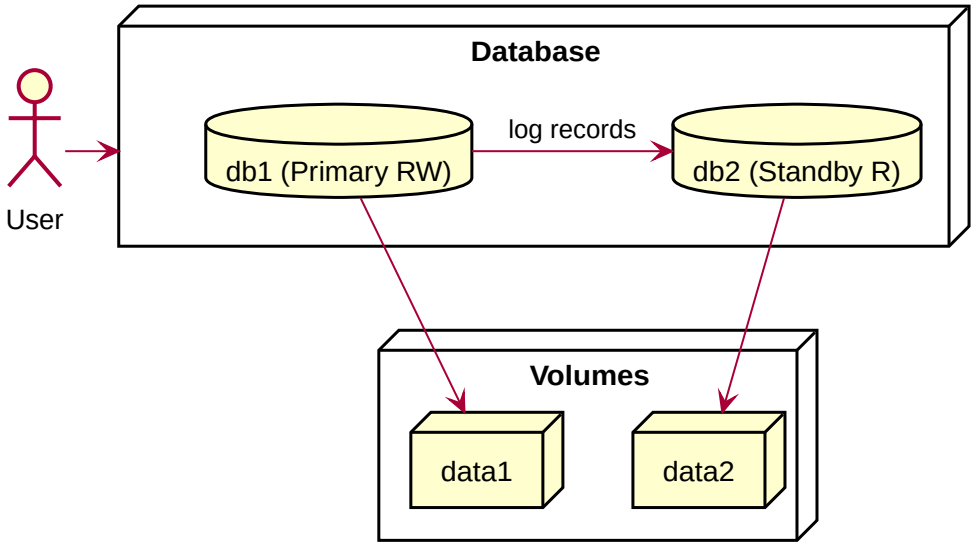


Figure 4. Hot / Warm Standby

Volume Sharing has the advantage of being easy to implement *if* you already have network storage. You can scale simply by increasing the amount of database instances (db3, db4, db5...). Each instance has full read / write access, making it easy to load balance. Despite that, reading / writing from network storage tends to be a bottleneck. Also, shared access to files and the issues that arise (file locking, etc.) are difficult problems. Your DBMS may not be able to handle them. Even if they can be handled, performance can be an issue. Lastly, this method puts all of your eggs in one basket with regard to where your data is stored. There may be no duplicates of the data depending on how you handle your network storage.

Hot / Warm Standby mode is typically already implemented by the DBMS. It is usually performed via *Log Shipping*, sending logs of all actions between a primary node and standby nodes. The standby nodes can keep up with transactions and be *promoted* in the event of an issue. Each node maintains a separate data volume. With this system, only the primary node is capable of performing write operations. If the standby node is a *hot* standby node it can perform reads as well. Fortunately, write operations tend to be less frequent that read operations, meaning you may see significant performance gains from scaling with this type of system.

What's the difference between a hot and warm standby? A hot standby can perform read operations while replicating the primary. This allows for load balancing to be performed. A warm standby simply keeps up with the primary so that it can be brought up in the event of a problem.

5.2. Implementation

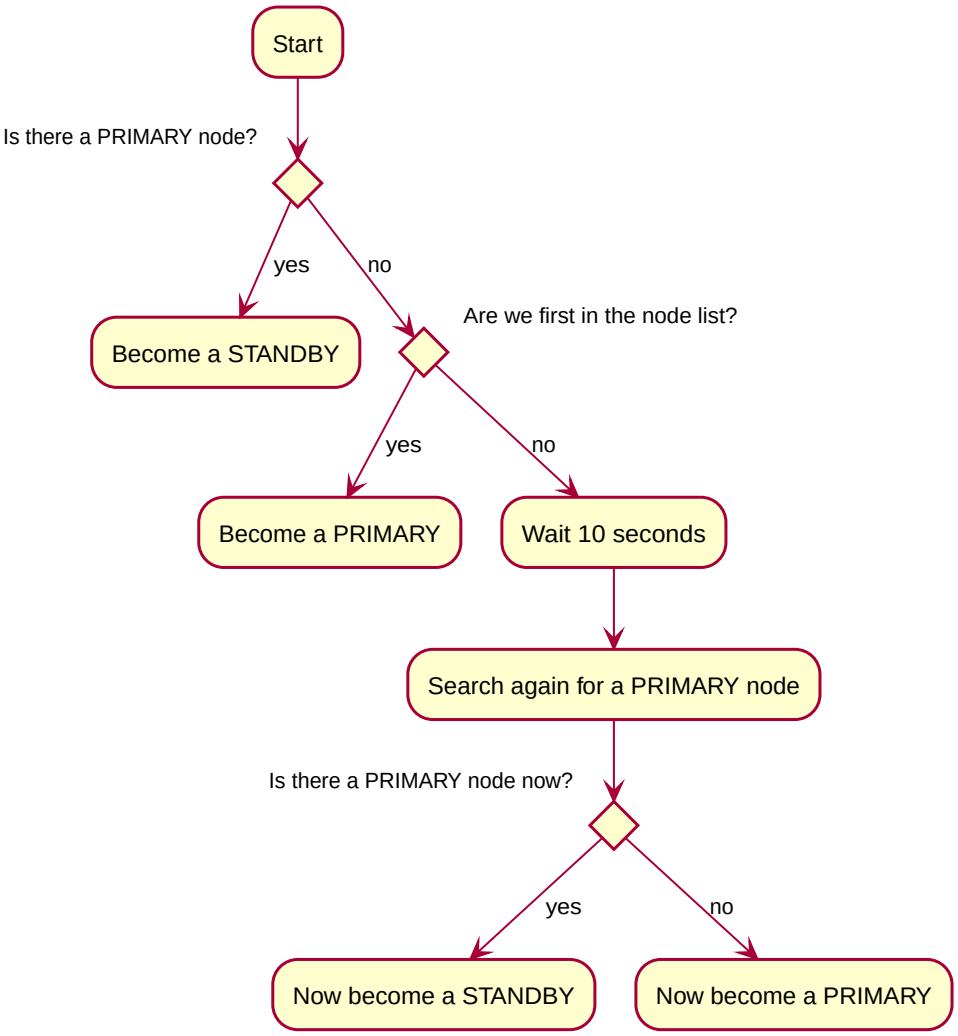
This example implements a Hot Standby in PostgreSQL. Actually implementing *replication* is largely handled for us by the DMBS. It is simply a matter of setting configuration variables, starting the databases are in the correct state, and bringing up the services in the right order. We can handle this in the `docker-entrypoint.sh` file that is used as the ENTRYPOINT for the container. In the official Docker image this script is responsible for setting up the database and running the user-specified command, `postgres` by default.

A basic Dockerfile that builds from this image and adds some extra needed utilities will be used:

replication-demo/db/Dockerfile

```
FROM postgres
RUN apt-get -y update && \
    apt-get -y install iputils-ping dnsutils
COPY docker-entrypoint.sh /usr/local/bin
```

Rather than create separate primary and standby images, this example employs one image that detects the status of the cluster on startup and creates either a primary or standby node accordingly. The logic for startup is as follows:



By passing a list of all nodes in an environment variable, we can create a bash function to see who is up and who is the primary when `docker-entrypoint.sh` is called at container creation:

replication-demo/db/docker-entrypoint.sh (excerpted)

```
function find_primary() {
    # Goes through all the nodes in POSTGRES_NODES and looks for one that is up
    # and is a PRIMARY
    echo "Looking for a primary node..."
    PRIMARY=""
    for NODE in $POSTGRES_NODES; do
        NODE_IP=$( dig +short $NODE )
        if [ -z $NODE_IP ] || [ $NODE_IP != $MY_IP ]; then
            # https://stackoverflow.com/questions/11231937/bash-ignoring-error-for-a-
particular-command
            EXIT_CODE=0
            pg_isready -h $NODE || EXIT_CODE=$?
            if [ $EXIT_CODE -eq 0 ]; then
                echo "$NODE:5432:postgres:postgres:$POSTGRES_PASSWORD" >> ~/.pgpass
                VALUE=$(psql -U postgres -t -h $NODE -d postgres -c "select
pg_is_in_recovery()")
                if [ $VALUE == "f" ]; then
                    echo "$NODE is primary node"
                    if [ ! -z $PRIMARY ]; then
                        echo "Two primary nodes detected! Exiting..."
                        exit 1
                    fi
                    PRIMARY=$NODE
                fi
            fi
        fi
    done
}
```

If a situation arises where there are two primaries on the network, this function will prevent a new db container from being created. No sense adding to the confusion.

Now all that is left is to configure either a primary or a standby node. The code to configure a primary node follows:

replication-demo/db/docker-entrypoint.sh (excerpted)

```
    echo "Configuring a PRIMARY instance..."

    if [ -s "$PGDATA/PG_VERSION" ]; then
        echo "Database already exists, NOT creating a new one"
    else
        echo "Creating a new database..."

        initdb --username=postgres --pwfile=<(echo "$POSTGRES_PASSWORD")

        # Start a temporary server listening on localhost
        pg_ctl -D "$PGDATA" -w start

        # Create a user for replication operations
        psql -v ON_ERROR_STOP=1 --username "$POSTGRES_USER" --dbname "$POSTGRES_DB" <<
EOSQL
            CREATE USER repuser REPLICATION LOGIN ENCRYPTED PASSWORD
'$POSTGRES_REPLICA_PASSWORD';
EOSQL

        # Stop the temporary server
        pg_ctl -D "$PGDATA" -m fast -w stop

        # Set up authentication parameters
        echo "host replication all all md5" >> $PGDATA/pg_hba.conf
        echo "host all all all md5" >> $PGDATA/pg_hba.conf
    fi

    # if, for some reason, a cold standby is being brought up as a primary
    # remove the standby.signal file
    if [ -f $PGDATA/standby.signal ]; then
        rm $PGDATA/standby.signal
    fi
```

The code is self-explanatory, but it should be noted that the actual building of the database, including auth configuration, and the create of a replication user is a rare occurrence. That should only happen once when the volume is initialized.

The code to configure a standby node is shown below:

replication-demo/db/docker-entrypoint.sh (excerpted)

```
    echo "Configuring a STANDBY instance..."

    # Set up our password so we can connect to replicate
    echo "$PRIMARY:5432:replication:repuser:$POSTGRES_REPLICA_PASSWORD" >>
/var/lib/postgresql/.pgpass

    # Clone the primary database
    rm -rf $PGDATA/*
    pg_basebackup -h $PRIMARY -D $PGDATA -U repuser -v -P -X stream
    chmod -R 700 $PGDATA

    # Add connection info
    cat << EOF >> $PGDATA/postgresql.conf
        primary_conninfo = 'host=$PRIMARY port=5432 user=repuser
password=$POSTGRES_REPLICA_PASSWORD'
EOF

    # Notify postgres that this is a standby server
    touch $PGDATA/standby.signal

    # Make sure there is a primary server and failover if there isn't
    monitor &
```

When a standby is brought up, any database on the volume is removed and the entire database from the primary is backed up You may want to revisit this design decision if your database becomes large. Connection info is also added to the end of `postgres.conf`. This does mean that if a standby is promoted the previous connect line will be synced to any new standbys that are brought up. This will result in an ever growing `postgres.conf` file. The monitor function will be covered in the next section.

You may notice that primaries and standbys have a very similar configuration. This is by design in PostgreSQL >= 12, to simplify the failover procedure.

Let's take a look at the relevant log messages of db1 and db2 when we bring them both up with `docker-compose up`:

```
db1 - Looking for a primary node...
db1 - db2:5432 - no response
db1 - Configuring a PRIMARY instance...
db2 - Looking for a primary node...
db2 - db1:5432 - no response
db2 - Giving the first node a 10s head start...
db2 - Looking for a primary node...
db2 - db1:5432 - accepting connections
db2 - db1 is primary node
db2 - Configuring a STANDBY instance...
```

As can be seen, db2 wasn't able to detect db1 at first but the additional 10s delay prevented a multiple-primary situation.

Having multiple primaries in a cluster is bad. So bad, that most solutions implement a STONITH policy. It is quite possibly the greatest acronym in all of technology.

Just because we have our database replicated on two volumes **does not** mean that we have backups. Those volumes are designed to be used as part of a running system and are not a reliable long-term solution. Create and implement a reliable backup plan as you would for any other database.

5.3. High Availability

Even though we now have a replicated database, it isn't doing us much good. If we want to make our database service highly available (HA) we will need to monitor for problems and promote a standby server if the primary server fails. This is referred to as *failover* and is often handled by a a separate component. In this simple example it is implemented in the `monitor` function shown below:

replication-demo/db/docker-entrypoint.sh (excerpted)

```
function monitor() {
    while true; do
        # spread out our checks to avoid the chance of two nodes promoting at
        # the same time
        WAIT=$((20 + $RANDOM % 10))
        sleep $WAIT
        find_primary
        if [ -z $PRIMARY ]; then
            echo "Can't find a primary failing over..."
            pg_ctl promote
            # we don't need to monitor any more if we are the primary
            exit
        fi
    done
}
```

This function is run in the background on every standby instance. The timeout is randomized to decrease the likelihood that two standby instances will promote themselves at exactly the same time. To better understand this, let's examine the **non-randomized** scenario shown in the following diagram:

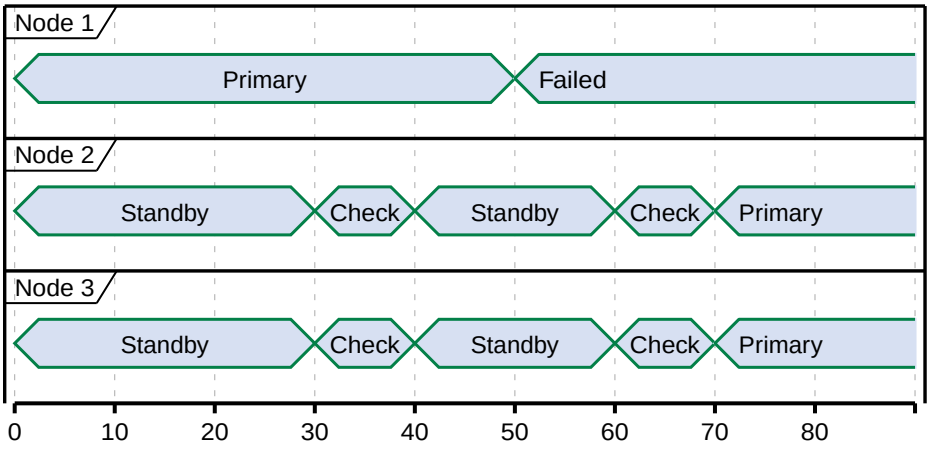


Figure 5. Dual Promotion

In the above scenario each standby node is checking to see that there is a primary node every 30s. The primary fails at 50s and *both* nodes check for a primary at exactly 60s. At that moment, neither node is a primary, no primary can be found, and they both begin the promotion process. This leaves the cluster with two primary nodes. This can be largely avoided by randomizing the check intervals.

Finally, lets simulate a failure and a recovery to see that our HA system is working. The following commands were executed and the Docker Compose logs were captured. Each command was run with about a minute pause between them:

1. `docker-compose up`
2. `docker-compose stop db1`
3. `docker-compose up db1`

The relevant, simplified log messages and descriptions follow:

```
db2 - Looking for a primary node...
db1 - Looking for a primary node...
db2 - db1:5432 - no response
db1 - db2:5432 - no response
db2 - Giving the first node a 10s head start...
db1 - Configuring a PRIMARY instance...
db2 - Looking for a primary node...
db2 - db1:5432 - accepting connections
db2 - db1 is primary node
db2 - Configuring a STANDBY instance...
```

Both db1 and db2 are brought up at the same time. db1 ends up being the primary.

```
db2 - Looking for a primary node...
db2 - db1:5432 - accepting connections
db2 - db1 is primary node
db2 - Looking for a primary node...
db2 - db1:5432 - accepting connections
db2 - db1 is primary node
```

db1 begins checking to make sure there is a primary node about every 30s.

```
db1 - LOG:  shutting down
db2 - Looking for a primary node...
db2 - db1:5432 - no response
db2 - Can't find a primary failing over...
db2 - server promoted
```

db1 is shut down. db2 performs its regularly scheduled check and self promotes because it cannot find a primary.

```
db1 - Looking for a primary node...
db1 - db2:5432 - accepting connections
db1 - db2 is primary node
db1 - Configuring a STANDBY instance...
db1 - Looking for a primary node...
db1 - db2:5432 - accepting connections
db1 - db2 is primary node
db1 - Looking for a primary node...
db1 - db2:5432 - accepting connections
db1 - db2 is primary node
```

db1 is brought back up, finds another primary and makes itself a standby. db1 begins checking to make sure the primary is available at regular intervals.

5.4. Load Balancing

Another advantage to replication is the ability to split the work among different nodes in the cluster. In our particular case, the primary node can handle any request, while the standby node can only handle read requests. Since we also control the application, we could create a connection for read requests and a separate connection for write requests.

Fortunately this can be accomplished by changing the connect string that is used in your application:

```
# get a read / write connection
psycopg.connect(
    database="example",
    host="db1,db2",
    user="postgres",
    password=password,
    target_session_attrs="read-write"
)

# get a read connection
psycopg.connect(
    database="example",
    host="db2,db1", # order should be randomized
    user="postgres",
    password=password,
    target_session_attrs="any"
)
```

The other option is to employ a proxy to forward requests, the most popular being HAProxy.

Load balancing is often referenced as a part of *horizontal scaling*. You can think of horizontal scaling as adding more instances to serve requests. *Vertical scaling* refers to making instances more powerful so that each individual one can serve more requests.

5.5. Questions

1. *What are some of the issues with a volume sharing database?*

2. *Why does our example have a startup delay? What could happen if we brought all of the nodes up at the same time?*

3. *What is the difference between high availability and load balancing?*

4. *What does a hot standby node do?*

5. *Our example starts up two nodes, but what would we have to change in our docker-compose.yml file if we wanted to start ten nodes? Is this congruent with the "Don't Repeat Yourself" (DRY) principle?*

Chapter 6. Kubernetes

6.1. Introduction

Kubernetes is a container orchestration system originally designed by Google. It is currently the most popular orchestration system and is notorious for being difficult to learn. Fortunately, we have already covered most of the concepts and the command syntax is similar to Docker.

A Kubernetes cluster is made up of nodes. Each node is capable of running pods, which in turn are running containers:

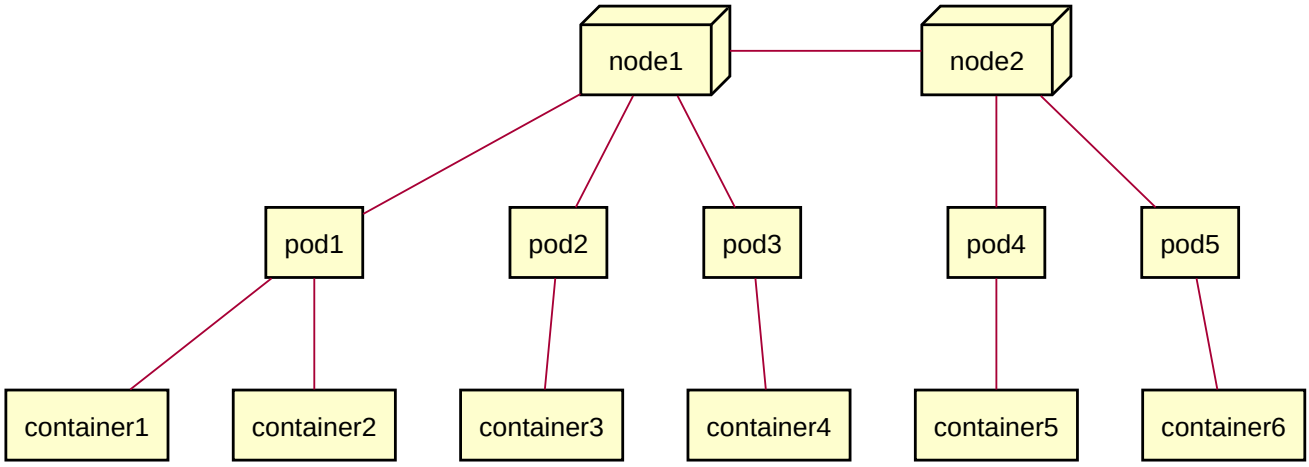


Figure 6. Kubernetes Cluster

Kubernetes is designed to solve the hard problems of multi-node deployment, replication, volume sharing, container communication, updates, roll-backs, service discovery and monitoring. It does this by defining objects and providing an API to interact with them. Some of the first objects we will be working with are:

Deployment

Defines how to handle the creation / maintenance of pods

Service

Defines what pods offer to the cluster and how it should be accessed

6.2. Minikube

For our purposes we will be using a single-node, local version of Kubernetes called minikube.

minikube acts as a single-node cluster by running Linux in a virtual machine on the host. It provides a several options for how the VM is run:

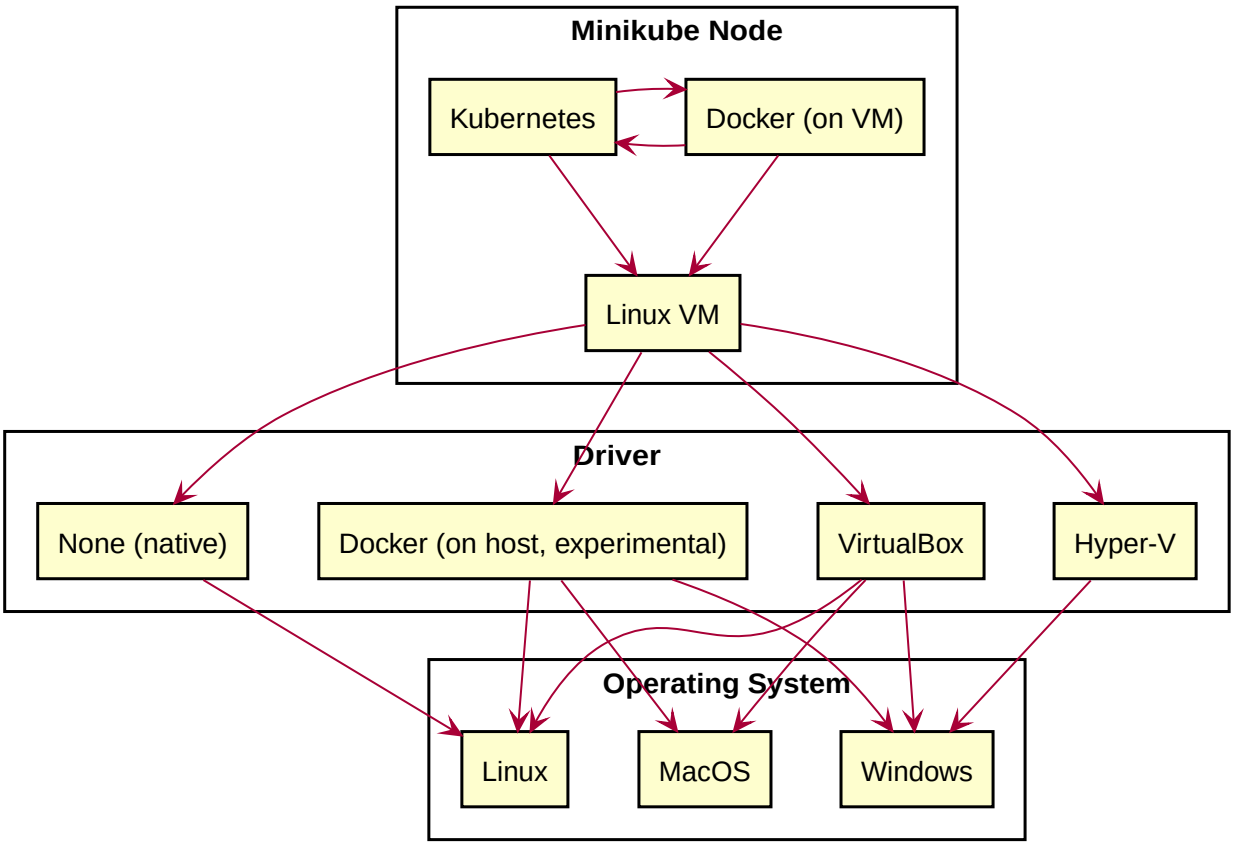


Figure 7. Minikube Architecture

Follow these directions to install minikube. There are a few virtualization options depending on the OS that you are running.

If you have Docker Desktop running, minikube defaults to the Docker driver. This driver is still experimental and may not work well. You can explicitly specify another driver when you start minikube with the `--driver=` option.

Hyper-V and VirtualBox were still mutually exclusive in Windows at the time of this writing. You will need to choose one or the other.

Hyper-V will require you to run your commands as an Administrator.

When you start minikube, you should see output similar to the following:

```
PS minikube-demo> minikube start --driver=hyperv
* minikube v1.9.0 on Microsoft Windows 10 Enterprise 10.0.18362 Build 18362
* Using the hyperv driver based on existing profile
* Retarting existing hyperv VM for "minikube" ...
* Preparing Kubernetes v1.18.0 on Docker 19.03.8 ...
* Enabling addons: default-storageclass, storage-provisioner
* Done! kubectl is now configured to use "minikube"
```

If you get errors due to low memory, close a few applications and try again. Typically you can restart the applications after minikube is running.

Kubernetes will attempt to pull all container images from a container repository by default. To avoid having to upload our images to a repository, we can set environment variables in our terminal so that we interact with the Docker daemon *inside* our minikube virtual machine.[3] Fortunately, minikube has the `docker-env` command to make this easier:

```
PS minikube-demo> minikube docker-env | Invoke-Expression
PS minikube-demo> docker ps
CONTAINER ID        IMAGE                  COMMAND                  CREATED
1bf91c2ca7df        4689081edb10           "/storage-provisioner"   7 minutes ago
a9866f5f5838        k8s.gcr.io/pause:3.2   "/pause"                 7 minutes ago
03a4f7f5e320        67da37a9a360           "/coredns -conf /etc…"   7 minutes ago
05061b993702        67da37a9a360           "/coredns -conf /etc…"   7 minutes ago
78977b068886        43940c34f24f           "/usr/local/bin/kube…"   7 minutes ago
b5312ba91086        k8s.gcr.io/pause:3.2   "/pause"                 7 minutes ago
968acc692934        k8s.gcr.io/pause:3.2   "/pause"                 7 minutes ago
0a72059bbe5f        k8s.gcr.io/pause:3.2   "/pause"                 7 minutes ago
d9c5aa3d43d0        303ce5db0e90           "etcd --advertise-cl…"   7 minutes ago
21b09398206f        74060cea7f70           "kube-apiserver --ad…"   7 minutes ago
313982f14a1c        d3e55153f52f           "kube-controller-man…"   7 minutes ago
35813d25f0bf        a31f78c7c8ce           "kube-scheduler --au…"   7 minutes ago
e6cdb564a306        k8s.gcr.io/pause:3.2   "/pause"                 7 minutes ago
b6bfe0e6f093        k8s.gcr.io/pause:3.2   "/pause"                 7 minutes ago
da47e560edab        k8s.gcr.io/pause:3.2   "/pause"                 7 minutes ago
3c599f97ecec        k8s.gcr.io/pause:3.2   "/pause"                 7 minutes ago
```

As can be seen from the output of the `docker ps` command, our Kubernetes cluster is made up of many containers running in a VM. If you ran `docker ps` without setting up the environment first, you would only see the containers you had running on your local Docker daemon (which may actually be running on a VM itself if you are using Docker Toolbox).

`minikube` commands will work in a new terminal, but if you want to build docker images and have them available on your Kubernetes cluster you will need to use minikube's docker-env command for each new terminal you open.

If you are experiencing errors with minikube, the first thing you should try is running `minikube delete` and `minikube start --driver=<your driver>`. This addresses the vast majority of issues by wiping all traces of the old VM, creating a new one, and starting fresh.

minikube includes a popular command line tool called kubectl. This is the command that we will be using for interacting with our Kubernetes cluster. To show that everything is working, lets create and build a basic Dockerfile that should print some output to standard out:

minikube-demo/Dockerfile

```
FROM alpine
ENTRYPOINT ["/bin/sh", "-c", "echo 'Hello from a Kubernetes log!'; sleep 30"]
```

```
PS minikube-demo> docker build -t k8s-example:v1 .
Sending build context to Docker daemon  2.048kB
Step 1/2 : FROM alpine
latest: Pulling from library/alpine
aad63a933944: Pull complete
Digest: sha256:b276d875eeed9c7d3f1cfa7edb06b22ed22b14219a7d67c52c56612330348239
Status: Downloaded newer image for alpine:latest
 ---> a187dde48cd2
Step 2/2 : ENTRYPOINT ["/bin/sh", "-c", "echo 'Hello from a Kubernetes log!'; sleep
30"]
 ---> Running in 96c1739d805e
Removing intermediate container 96c1739d805e
 ---> 7b9898952ce0
Successfully built 7b9898952ce0
Successfully tagged k8s-example:v1
```

We built our image with a tag *and* a version. You need the tag so you can reference it from Kubernetes. If you don't specify a version Kubernetes will try to pull the `latest` from a repository.

Now we'll create a deployment, which by default will make one pod that runs your image. We also run the `get deployment` and `get pod` commands so we can see the outcome. Lastly, we will inspect the logs for the pod that was created.

```
PS minikube-demo> kubectl create deployment k8s-example --image=k8s-example:v1
deployment.apps/k8s-example created
PS minikube-demo> kubectl get deployment
NAME          READY   UP-TO-DATE   AVAILABLE   AGE
k8s-example   1/1     1            1           7s
PS minikube-demo> kubectl get pod
NAME                           READY   STATUS    RESTARTS   AGE
k8s-example-5787cd97dc-ft2cr   1/1     Running   0          12s
PS minikube-demo> kubectl logs k8s-example-5787cd97dc-ft2cr
Hello from a Kubernetes log!
```

Our image is up and running, but remember that after 30 seconds it should exit. As an orchestration system, Kubernetes defaults to restarting pods that have stopped. Lets wait a while (14 minutes to be exact) and then execute the `get pod` command again:

```
PS minikube-demo> kubectl get pod
NAME                           READY   STATUS             RESTARTS   AGE
k8s-example-5787cd97dc-ft2cr   0/1     CrashLoopBackOff   6          14m
```

Kubernetes has restarted our pod six times now. In fact, it restarted it so much that it is now waiting before trying again (`CrashLoopBackOff`). You now have a minikube single-node Kubernetes cluster running on your local machine. You can build custom Docker images and have them run on your cluster.

To bring everything down, use the `delete deployment` command:

```
PS minikube-demo> kubectl delete deployment k8s-example
deployment.apps "k8s-example" deleted
PS minikube-demo> kubectl get pod
No resources found in default namespace.
```

6.3. Debugging

The official Kubernetes documentation has an excellent article on debugging services. What follows are some tips they may help reinforce or fill-in-the-blanks for topics in the article.

Two `kubectl` commands are especially useful for finding out more information about an object:

`kubectl get`
: gets brief information about an object or all of the objects of that type

`kubectl describe`
: gets more information about an object

Let's take a look:

```
PS minikube-demo> kubectl get pod
NAME                           READY   STATUS    RESTARTS   AGE
k8s-example-5787cd97dc-fbrxl   1/1     Running   0          33s
PS minikube-demo> kubectl describe pod k8s-example-5787cd97dc-fbrxl
Name:         k8s-example-5787cd97dc-fbrxl
Namespace:    default
Priority:     0
Node:         minikube/172.17.0.2
Start Time:   Mon, 13 Apr 2020 18:22:39 -0400
Labels:       app=k8s-example
              pod-template-hash=5787cd97dc
Annotations:  <none>
Status:       Running
IP:           172.18.0.3
IPs:
  IP:           172.18.0.3
Controlled By:  ReplicaSet/k8s-example-5787cd97dc
Containers:
  k8s-example:
    Container ID:
docker://f22a1be8401f256c42c8c8ad82cf6757bc9e34ec7ae1fe0c4329fff57ff09bcb
    Image:          k8s-example:v1
    Image ID:
docker://sha256:374b52c1385d25269a05e9542e65690fe9dc00146b869580db8ab51b5027096a
    Port:           <none>
    Host Port:      <none>
    State:          Running
      Started:      Mon, 13 Apr 2020 18:23:37 -0400
    Last State:     Terminated
      Reason:       Completed
      Exit Code:    0
      Started:      Mon, 13 Apr 2020 18:23:06 -0400
      Finished:     Mon, 13 Apr 2020 18:23:36 -0400
    Ready:          True
    Restart Count:  1
    Environment:    <none>
    Mounts:
      /var/run/secrets/kubernetes.io/serviceaccount from default-token-5mgft (ro)
Conditions:
  Type              Status
  Initialized       True
  Ready             True
  ContainersReady   True
  PodScheduled      True
Volumes:
  default-token-5mgft:
    Type:        Secret (a volume populated by a Secret)
    SecretName:  default-token-5mgft
    Optional:    false
QoS Class:       BestEffort
Node-Selectors:  <none>
```

```
Tolerations:     node.kubernetes.io/not-ready:NoExecute for 300s
                 node.kubernetes.io/unreachable:NoExecute for 300s
Events:
  Type     Reason     Age                From               Message
  ----     ------     ----               ----               -------
  Normal   Scheduled  <unknown>          default-scheduler  Successfully assigned
default/k8s-example-5787cd97dc-fbrxl to minikube
  Normal   BackOff    62s                kubelet, minikube  Back-off pulling image
"k8s-example:v1"
  Warning  Failed     62s                kubelet, minikube  Error: ImagePullBackOff
  Normal   Pulling    50s (x2 over 62s)  kubelet, minikube  Pulling image "k8s-
example:v1"
  Warning  Failed     50s (x2 over 62s)  kubelet, minikube  Failed to pull image "k8s-
example:v1": rpc error: code = Unknown desc = Error
 response from daemon: pull access denied for k8s-example, repository does not exist
or may require 'docker login': denied: requested acc
ess to the resource is denied
  Warning  Failed     50s (x2 over 62s)  kubelet, minikube  Error: ErrImagePull
  Normal   Pulled     5s (x2 over 36s)   kubelet, minikube  Container image "k8s-
example:v1" already present on machine
  Normal   Created    5s (x2 over 36s)   kubelet, minikube  Created container k8s-
example
  Normal   Started    5s (x2 over 36s)   kubelet, minikube  Started container k8s-
example
```

As you can see there, is lots of useful information here including a full history of events that have occured.

Seeing `ErrImagePull` or `ImagePullBackOff` in the status section of `kubectl get pod` is a common problem. Check to make sure you've set up your environment correctly to use the Docker daemon *in* minikube and then try the `docker pull <image>` command yourself. If Docker can't pull it check to see that you are connected to the network and if it's a custom image check to see that you built it. `docker images` will show you all of the images minikube can access.

Many of the same techniques used for debugging Docker images can be used for debugging Kubernetes objects. For example you can execute interactive commands (including a shell) on running docker images with `kubectl exec -it`:

```
PS minikube-demo> kubectl exec -it k8s-example-5787cd97dc-7j6cc -- /bin/sh
/ # ps ax
PID   USER     TIME  COMMAND
    1 root      0:00 sleep 30
   11 root      0:00 /bin/sh
   16 root      0:00 ps ax
/ # ls
bin    dev    etc    home   lib    media  mnt    opt    proc   root   run    sbin
srv    sys    tmp    usr    var
/ # exit
```

If, for some reason, an image does not start up, you can replace the ENTRYPOINT of the Dockerfile from within the `template` definition of a **Deployment**. By replacing it with something you know will work, you can then execute an interactive shell in the container to see what is going on. The following is an example of executing the sleep command, assuring that the pod will run for at least an hour:

```
kind: Deployment
metadata:
  name: db-rw
  labels:
    app: db-rw
spec:
  replicas: 1
  selector:
    matchLabels:
      app: db-rw
  template:
    metadata:
      labels:
        app: db-rw
    spec:
      containers:
        - name: db-rw
          image: postgres
          env:
            - name: POSTGRES_PASSWORD
              value: "changeme"
            - name: POSTGRES_REPLICA_PASSWORD
              value: "changeme"
          command: ["bash", "-c", "sleep 3600"]
```

Sometimes your objects are applied, but no pods start up. This may mean that Kubernetes started your **Deployment** or **StatefulSet** which created a **ReplicaSet**, but the **ReplicaSet** was unable to start your pods. Try running `kubectl get replicaset` to find which **ReplicaSet** is running and then `kubectl describe replicaset <replicaset_name>` where <replicaset_name> is the name of your **ReplicaSet**.

Finally, you may find yourself in a situation where you need to run `kubectl` from within a pod. This can be helpful for sorting out role based access control issues, namely "how can this pod interact with the Kubernetes API?" This guide is a great resource. It largely boils down to:

1. From within the pod (`kubectl exec -it <pod-name> -- bash`) install curl: `apt-get install curl`.
2. Download `kubectl`:
   a. `VERSION=curl https://storage.googleapis.com/kubernetes-release/release/stable.txt`
   b. `curl -LO https://storage.googleapis.com/kubernetes-release/release/$VERSION/bin/linux/amd64/kubectl`
3. Make it executable and install it:
   a. `chmod +x ./kubectl`
   b. `mv ./kubectl /usr/local/bin/`

6.4. Conclusion

Hopefully you can see the benefit of working with an orchestration framework. While it may be daunting at first to learn all of the objects and to use new, unfamiliar commands, Kubernetes does provide a lot of options for someone looking to deploy scalable applications. Kubernetes has emerged as the de facto standard and knowing how to use it is a *very* marketable skill.

6.5. Questions

1. *What is the role of a pod in Kubernetes?*

2. *What does a **Deployment** do?*

3. *What is minikube used for and what platforms can it run on?*

4. *If you were given an image to deploy on Kubernetes and it continually failed to start, what steps would you take to figure out what was going wrong?*

[3] See this great blog post for more details

Chapter 7. Database in Kubernetes

7.1. Introduction

In this example we implement a primary / standby replication setup for PostgreSQL. Two **Services** will be provided: one for read/write requests and another exclusively for read requests. We will try to use Kubernetes to handle the initialization and monitoring functions that had to be done by hand in previously.

Rather than passing command line options to the `kubectl` command, we will be defining what we create in a YAML file: `example-final/db-k8s.yml`. kubectl can load object definitions from YAML files with the `apply` command.

Now let's look at the Kubernetes objects we are defining:

7.2. PersistentVolumeClaims

A **PersistentVolumeClaim** lets the cluster know that you are expecting certain storage resources. In this case, we are looking for a place to store our primary database files. This claim will be fulfilled by a **StorageClass** that is built into minikube. As far as we are concerned, we just have to tell it what we want and it will make it happen.[4].

example-final/db-k8s.yml (excerpted)

```
---
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: db-primary-pv-claim
spec:
  accessModes:
    - ReadWriteOnce
  resources:
    requests:
      storage: 512M
```

This claim will be used by our *one* `db-rw` pod, so we don't have to worry about shared access. The supported accessModes are:

- **ReadWriteOnce** - can be mounted read / write by only one pod
- **ReadOnlyMany** – can be mounted read-only by many pods
- **ReadWriteMany** – can be mounted as read-write by many pods

7.3. Services

A **Service** exposes an application on a group of pods. In our case we will be providing two

Services: a `db-rw` **Service** which connects to our primary PostgreSQL instance and a `db-r` **Service** which connects to our standby PostgreSQL instances. Unlike Docker Compose, even on our internal network we have to explicitly state which ports we make available.

example-final/db-k8s.yml (excerpted)

```
---
apiVersion: v1
kind: Service
metadata:
  name: db-rw
  labels:
    app: db-rw
spec:
  selector:
    app: db-rw
  ports:
    - protocol: TCP
      port: 5432

---
apiVersion: v1
kind: Service
metadata:
  name: db-r
  labels:
    app: db-r
spec:
  selector:
    app: db-r
  ports:
    - protocol: TCP
      port: 5432
```

The `selector` field above defines how a **Service** knows which pods to utilize. In our case all pods with the app label `db-r` are used by the `db-r` **Service** and a similar rule is applied to the `db-rw` **Service**. Both services accept incoming connections on port 5432 and route those connections to 5432. In the case where there are multiple pods in a **Service** a load balancing scheme is used by default.

From a service discovery perspective, Kubernetes **Services** make things easier. If you want to connect to a read-only database instance all you have to do is use the hostname `db-r`. Similarly, if you want to connect to a read-write database, use the hostname `db-rw`. The DNS resolution, load-balancing proxy, and routing are set up automatically.

7.4. Deployments

A **Deployment** tells Kubernetes how to create and monitor pods. The bulk of our work will be done in the `db-r` and `db-rw` **Deployments**. Fortunately we have already covered the logic of what needs to

happen previously, so let's jump right in and take a look at the **Deployment** for our primary PostgreSQL instance:

example-final/db-k8s.yml (excerpted)

```
---
apiVersion: apps/v1
kind: Deployment
metadata:
  name: db-rw
  labels:
    app: db-rw
spec:
  replicas: 1
  selector:
    matchLabels:
      app: db-rw
  template:
    metadata:
      labels:
        app: db-rw
    spec:
      containers:
        - name: db-rw
          image: postgres:12.2
          env:
            - name: POSTGRES_PASSWORD
              value: "changeme"
            - name: POSTGRES_REPLICA_PASSWORD
              value: "changeme"
          command:
            - bash
            - "-c"
            - |
              set -ex

              if [ -s "/var/lib/postgresql/data/PG_VERSION" ]; then
                echo "Database already exists, not creating a new one."
              else
                rm -rf /var/lib/postgresql/data/*
                chown postgres /var/lib/postgresql/data

                su -c "initdb --username=postgres --pwfile=<(echo
\"$POSTGRES_PASSWORD\")" postgres

                # Start a temporary server listening on localhost
                su -c "pg_ctl -D /var/lib/postgresql/data -w start" postgres

                # Create a user for replication operations and initialize our
                # example database
                psql -v ON_ERROR_STOP=1 --username postgres --dbname postgres <<EOF
```

```
                  CREATE USER repuser REPLICATION LOGIN ENCRYPTED PASSWORD
'$POSTGRES_REPLICA_PASSWORD';
                  CREATE DATABASE example;
                  \c example
                  CREATE TABLE users(
                    email VARCHAR(255) PRIMARY KEY,
                    hash VARCHAR(255) NOT NULL
                  );
              EOF
              # ^ this EOF has to be in line with the YAML scalar block

                # Stop the temporary server
                su -c "pg_ctl -D /var/lib/postgresql/data -m fast -w stop" postgres

                # Set up authentication parameters
                echo "host replication all all md5" >>
/var/lib/postgresql/data/pg_hba.conf
                echo "host all all all md5" >> /var/lib/postgresql/data/pg_hba.conf
              fi

              # Now run the server
              su -c postgres postgres
          volumeMounts:
            - name: db-primary-storage
              mountPath: /var/lib/postgresql/data
      volumes:
        - name: db-primary-storage
          persistentVolumeClaim:
            claimName: db-primary-pv-claim
```

Ths **Deployment** tells Kubernetes to maintain one `replica` of the pod defined in the `template` section. The `containers` section is a list of one container that uses the `postgres` image from Docker Hub and overrides the ENTRYPOINT of that Dockerfile (this is `command` in Kubernetespeak). Our script is taken almost line-for-line from our previous example. Lastly, this deployment makes use of our **PersistentVolumeClaim** defined previously and mounts it in `/var/lib/postgresql/data`.

Let's take a look at the **Deployment** for our standby PostgreSQL instances:

example-final/db-k8s.yml (excerpted)

```
---
apiVersion: apps/v1
kind: Deployment
metadata:
  name: db-r
  labels:
    app: db-r
spec:
  replicas: 2
  selector:
    matchLabels:
```

```
      app: db-r
  template:
    metadata:
      labels:
        app: db-r
    spec:
      containers:
        - name: db-r
          image: postgres:12.2
          env:
            - name: POSTGRES_REPLICA_PASSWORD
              value: "changeme"
          command:
            - bash
            - "-c"
            - |
              set -ex

              # Set up our password in .pgpass so we can connect to replicate
              # without a prompt
              echo "db-rw:5432:replication:repuser:$POSTGRES_REPLICA_PASSWORD" >>
/var/lib/postgresql/.pgpass
              chown postgres /var/lib/postgresql/.pgpass
              chmod 600 /var/lib/postgresql/.pgpass

              # we may start before their are WALs, so we need to make this directory
              mkdir -p /var/lib/postgresql/data/pg_wal
              chown postgres /var/lib/postgresql/data/pg_wal

              # Clone the database from db-rw
              rm -rf /var/lib/postgresql/data/*
              chown postgres /var/lib/postgresql/data
              chmod -R 700 /var/lib/postgresql/data
              su -c "pg_basebackup -h db-rw -D /var/lib/postgresql/data -U repuser -w
-v -P -X stream" postgres

              # Add connection info
              cat << EOF >> /var/lib/postgresql/data/postgresql.conf
                primary_conninfo = 'host=db-rw port=5432 user=repuser
password=$POSTGRES_REPLICA_PASSWORD'
              EOF

              # Notify postgres that this is a standby server
              touch /var/lib/postgresql/data/standby.signal

              # Now run the server
              su -c postgres postgres
```

This **Deployment** stands up two replicas. Each replica clones the primary database (using the hostname `db-rw` provided by our `db-rw` **Service**) and then acts as a hot standby. A

PersistentVolumeClaim is *not* used, meaning if push came to shove, we may not be able to easily recover the database from one of these containers.

Notice that neither **Deployment** has to search for the primary or monitor the other instances. Kubernetes handles this for us. In fact, the standbys don't even have to wait for the primary to be up. If they can't clone the database, they will fail and Kubernetes will restart them until they work.

Much of the hard work of our earlier example is now handled for us by a *proper* orchestration framework.

7.5. Running the Example

Let's take a look at the example in action:

```
PS example-final> kubectl apply -f .\db-k8s.yml
persistentvolumeclaim/db-primary-pv-claim created
service/db-rw created
service/db-r created
deployment.apps/db-rw created
deployment.apps/db-r created
```

The `kubectl apply -f` command can be used to bring up all of the objects defined in a file. It should also be noted that it can work with an entire directory of files, allowing for separation of logical segments, unlike a `docker-compose.yml` file.

```
PS example-final> kubectl get pod
NAME                     READY   STATUS    RESTARTS   AGE
db-r-54d9bc6496-cjhn8    1/1     Running   1          2m31s
db-r-54d9bc6496-pg8b2    1/1     Running   0          2m31s
db-rw-6fd7767ddd-g6kvj   1/1     Running   0          2m31s
```

`kubectl get pod` show you all of the pods that are currently running. All of the pods in our deployment are now up. They are given hash codes for the second part of their name to keep them unique. You may notice that `db-r-54d9bc6496-cjhn8` had to be restarted once. It probably came up before `db-rw-6fd7767ddd-g6kvj` was ready to have its database cloned.

Using the command `kubectl logs` we can get the logs for a pod. Let's take a look at our primary PostgreSQL instance:

```
PS example-final> kubectl logs db-rw-6fd7767ddd-g6kvj
+ '[' -s /var/lib/postgresql/PG_VERSION ']'
+ rm -rf '/var/lib/postgresql/data/*'
+ chown postgres /var/lib/postgresql/data
+ su -c 'initdb --username=postgres --pwfile=<(echo "changeme")' postgres
The files belonging to this database system will be owned by user "postgres".
This user must also own the server process.
```

```
The database cluster will be initialized with locale "en_US.utf8".
The default database encoding has accordingly been set to "UTF8".
The default text search configuration will be set to "english".

Data page checksums are disabled.

fixing permissions on existing directory /var/lib/postgresql/data ... ok
creating subdirectories ... ok
selecting dynamic shared memory implementation ... posix
selecting default max_connections ... 100
selecting default shared_buffers ... 128MB
selecting default time zone ... Etc/UTC
creating configuration files ... ok
running bootstrap script ... ok
performing post-bootstrap initialization ... ok
initdb: warning: enabling "trust" authentication for local connections
You can change this by editing pg_hba.conf or using the option -A, or
--auth-local and --auth-host, the next time you run initdb.
syncing data to disk ... ok


Success. You can now start the database server using:

    pg_ctl -D /var/lib/postgresql/data -l logfile start

+ su -c 'pg_ctl -D /var/lib/postgresql/data -w start' postgres
waiting for server to start....2020-04-06 01:34:45.086 UTC [27] LOG:  starting
PostgreSQL 12.2 (Debian 12.2-2.pgdg100+1) on x86_64
-pc-linux-gnu, compiled by gcc (Debian 8.3.0-6) 8.3.0, 64-bit
2020-04-06 01:34:45.086 UTC [27] LOG:  listening on IPv4 address "0.0.0.0", port 5432
2020-04-06 01:34:45.086 UTC [27] LOG:  listening on IPv6 address "::", port 5432
2020-04-06 01:34:45.091 UTC [27] LOG:  listening on Unix socket
"/var/run/postgresql/.s.PGSQL.5432"
2020-04-06 01:34:45.104 UTC [28] LOG:  database system was shut down at 2020-04-06
01:34:44 UTC
2020-04-06 01:34:45.109 UTC [27] LOG:  database system is ready to accept connections
 done
server started
+ psql -v ON_ERROR_STOP=1 --username postgres --dbname postgres
CREATE ROLE
+ su -c 'pg_ctl -D /var/lib/postgresql/data -m fast -w stop' postgres
waiting for server to shut down....2020-04-06 01:34:45.238 UTC [27] LOG:  received
fast shutdown request
2020-04-06 01:34:45.242 UTC [27] LOG:  aborting any active transactions
2020-04-06 01:34:45.244 UTC [27] LOG:  background worker "logical replication
launcher" (PID 34) exited with exit code 1
2020-04-06 01:34:45.244 UTC [29] LOG:  shutting down
2020-04-06 01:34:45.275 UTC [27] LOG:  database system is shut down
 done
server stopped
+ echo 'host replication all all md5'
```

```
+ echo 'host all all all md5'
+ su -c postgres postgres
2020-04-06 01:34:45.362 UTC [46] LOG:  starting PostgreSQL 12.2 (Debian 12.2-
2.pgdg100+1) on x86_64-pc-linux-gnu, compiled by gcc
(Debian 8.3.0-6) 8.3.0, 64-bit
2020-04-06 01:34:45.363 UTC [46] LOG:  listening on IPv4 address "0.0.0.0", port 5432
2020-04-06 01:34:45.363 UTC [46] LOG:  listening on IPv6 address "::", port 5432
2020-04-06 01:34:45.368 UTC [46] LOG:  listening on Unix socket
"/var/run/postgresql/.s.PGSQL.5432"
2020-04-06 01:34:45.383 UTC [47] LOG:  database system was shut down at 2020-04-06
01:34:45 UTC
2020-04-06 01:34:45.388 UTC [46] LOG:  database system is ready to accept connections
```

Just as before, the primary node initialized a database, set up a replication user, and started `postgres`.

Let's take a look at one of the standby nodes:

```
PS example-final> kubectl logs db-r-54d9bc6496-cjhn8
+ echo db-rw:5432:replication:repuser:changeme
+ chown postgres /var/lib/postgresql/.pgpass
+ chmod 600 /var/lib/postgresql/.pgpass
+ mkdir -p /var/lib/postgresql/data/pg_wal
+ chown postgres /var/lib/postgresql/data/pg_wal
+ rm -rf /var/lib/postgresql/data/pg_wal
+ chown postgres /var/lib/postgresql/data
+ chmod -R 700 /var/lib/postgresql/data
+ su -c 'pg_basebackup -h db-rw -D /var/lib/postgresql/data -U repuser -w -v -P -X
stream' postgres
pg_basebackup: initiating base backup, waiting for checkpoint to complete
pg_basebackup: checkpoint completed
pg_basebackup: write-ahead log start point: 0/3000028 on timeline 1
pg_basebackup: starting background WAL receiver
pg_basebackup: created temporary replication slot "pg_basebackup_57"
    0/24554 kB (0%), 0/1 tablespace (...lib/postgresql/data/backup_label)
24564/24564 kB (100%), 0/1 tablespace (...ostgresql/data/global/pg_control)
24564/24564 kB (100%), 1/1 tablespace
pg_basebackup: write-ahead log end point: 0/3000100
pg_basebackup: waiting for background process to finish streaming ...
pg_basebackup: syncing data to disk ...
pg_basebackup: base backup completed
+ cat
+ touch /var/lib/postgresql/data/standby.signal
+ su -c postgres postgres
2020-04-06 01:34:47.283 UTC [18] LOG:  starting PostgreSQL 12.2 (Debian 12.2-
2.pgdg100+1) on x86_64-pc-linux-gnu, compiled by gcc
(Debian 8.3.0-6) 8.3.0, 64-bit
2020-04-06 01:34:47.283 UTC [18] LOG:  listening on IPv4 address "0.0.0.0", port 5432
2020-04-06 01:34:47.283 UTC [18] LOG:  listening on IPv6 address "::", port 5432
2020-04-06 01:34:47.289 UTC [18] LOG:  listening on Unix socket
"/var/run/postgresql/.s.PGSQL.5432"
2020-04-06 01:34:47.304 UTC [19] LOG:  database system was interrupted; last known up
at 2020-04-06 01:34:46 UTC
2020-04-06 01:34:47.442 UTC [19] LOG:  entering standby mode
2020-04-06 01:34:47.446 UTC [19] LOG:  redo starts at 0/3000028
2020-04-06 01:34:47.449 UTC [19] LOG:  consistent recovery state reached at 0/3000100
2020-04-06 01:34:47.449 UTC [18] LOG:  database system is ready to accept read only
connections
2020-04-06 01:34:47.455 UTC [23] LOG:  started streaming WAL from primary at 0/4000000
on timeline 1
```

This standby backed up the primary database and started streaming logs from the primary.

The `kubectl exec` command lets you execute a command on a running pod. Lets use this to run psql on one of the standbys, connect to the primary, create a table, and then check to see if it shows up on the standbys:

```
PS example-final> kubectl exec -it db-r-54d9bc6496-pg8b2 -- bash
root@db-r-54d9bc6496-pg8b2:/# psql -h db-rw -U postgres
Password for user postgres:
psql (12.2 (Debian 12.2-2.pgdg100+1))
Type "help" for help.

postgres=# \dt
Did not find any relations.
postgres=# CREATE TABLE test(test_column INTEGER);
CREATE TABLE
postgres=# \dt
        List of relations
 Schema | Name | Type  |  Owner
--------+------+-------+----------
 public | test | table | postgres
(1 row)

postgres=# \q
root@db-r-54d9bc6496-pg8b2:/# psql -h db-r -U postgres
Password for user postgres:
psql (12.2 (Debian 12.2-2.pgdg100+1))
Type "help" for help.

postgres=# \dt
        List of relations
 Schema | Name | Type  |  Owner
--------+------+-------+----------
 public | test | table | postgres
(1 row)

postgres=# \q
```

Sure enough, anything we create on the primary (which we access by resolving the name `db-rw` shows up on the standby. Now lets try performing a write operation on a standby:

```
root@db-r-54d9bc6496-pg8b2:/# psql -h db-r -U postgres
Password for user postgres:
psql (12.2 (Debian 12.2-2.pgdg100+1))
Type "help" for help.

postgres=# CREATE TABLE test2(test_column INTEGER);
ERROR:  cannot execute CREATE TABLE in a read-only transaction
```

It fails, as it should. Lets take a look at how **Services** glue all of this together with the `kubectl get service` command:

```
PS example-final> kubectl get service
NAME         TYPE        CLUSTER-IP      EXTERNAL-IP   PORT(S)    AGE
db-r         ClusterIP   10.106.33.23    <none>        5432/TCP   41m
db-rw        ClusterIP   10.99.113.228   <none>        5432/TCP   41m
kubernetes   ClusterIP   10.96.0.1       <none>        443/TCP    35h
```

`db-r` and `db-rw` are **Services**, so if a pod tries to resolve on of those names, they will get the CLUSTER-IP (10.106.33.23 and 10.99.113.228 respectively). That ClusterIP is a proxy that will forward their request to a pod that can handle it. This allows for load-balancing and high availability.

On the subject of HA, the last thing we have to check is that pods will be automatically restarted. Let's do something bad to one of our pods with the `kubectl exec` command:

```
PS example-final> kubectl exec -it db-rw-6fd7767ddd-g6kvj -- bash
root@db-rw-6fd7767ddd-g6kvj:/# killall5 -9
command terminated with exit code 137
PS C:\Users\rxt1077\it490\example-final> kubectl get pods
NAME                     READY   STATUS    RESTARTS   AGE
db-r-54d9bc6496-cjhn8    1/1     Running   1          48m
db-r-54d9bc6496-pg8b2    1/1     Running   0          48m
db-rw-6fd7767ddd-g6kvj   1/1     Running   1          48m
```

`killall5 -9` will send a kill signal to all processes on the pod, causing it to shut down. Kubernetes brought it back up again as evidenced by RESTARTS being equal to one. While the primary is restarting you will lose write access, but the standbys will continue to provide read access. Once it is back up (a matter of seconds), everything should be functioning as normal.

7.6. Conclusion

Using Kubernetes we were able to quickly build a HA PostgreSQL cluster. This is just the tip of the iceberg for what Kubernetes supports and as you learn more about it you should be able to revisit and improve this implementation.

7.7. Questions

1. *A systems architect was using a stock Docker Hub image with a custom ENTRYPOINT point script she had designed. This required a Dockerfile, BASH script, and a directory to store them. When she migrated to Kubernetes she was able to do this all in one YAML file. Describe how this is possible.*

2. *Why are **Services** essential to replication?*

3. *Why do we define two **Deployments** for our example?*

4. *How can our database deployment be improved?*

5. *Compare and contrast Kubernetes PersistentVolumeClaims with Docker compose named volumes.*

[4] The fascinating details of how this works are revealed in this blog post.

Chapter 8. Messaging in Kubernetes

In this section we will build a RabbitMQ cluster in Kubernetes to support our application.

8.1. Resources

Before we get started, there are some excellent resources for what we are going to be covering. You should read / explore *all* of the following resources:

- RabbitMQ Clustering Guide
- RabbitMQ Cluster Formation and Peer Discovery
- RabbitMQ Documentation on Docker Hub
- Deploy RabbitMQ on Kubernetes with the Kubernetes Peer Discovery Plugin

8.2. RabbitMQ

RabbitMQ is built on Erlang/OTP, a platform designed in the telecom industry. Given the nature of that industry, Erlang/OTP was designed to be highly scalable and have strong concurrency support. In other words, it is the perfect platform for building non-hierarchical, distributed applications. To give you an example that you may be familiar with, Whatsapp runs on Erlang/OTP and it handles about two million connected users per server.

All nodes in a RabbitMQ cluster are equal peers, there is no primary / standby structure like we used in the database example. The RAFT consensus algorithm is used to make decisions for the cluster and as such, it is highly recommended that the number of nodes be odd. Given the amount of traffic and the need for quick communication, clustering is designed to function at the LAN level, not the WAN level.

Messages in queues are **not** replicated by default, although that can be turned on. For us, this only really matters for the `incoming` queue as our other queues are exclusive. Any node can route requests through the node that happens to contain the `incoming` queue and given the short nature of our connections this should function just fine. It is also worth nothing that when a node joins a cluster, its state is reset. Once again, given the nature of our connections this shouldn't have much of an impact on our application.

It is recommended that all nodes run the same version of Erlang/OTP. This should be easy for us since our nodes will be built from the same image. Nodes also need to have the same Erlang cookie (shared secret) so they can communicate with each other securely. This can be passed via the `RABBITMQ_ERLANG_COOKIE` environment variable.

8.3. Kubernetes

RabbitMQ has a peer discovery plugin for Kubernetes that is included with its base image. It just needs to be enabled. RabbitMQ also provides a repository that demonstrates how to use it. This example will implement something similar, but before we do, we need to go over some new Kubernetes objects.

Our example will make use of **StatefulSets**, which are similar to **Deployments** in that they use a template to build pods. **StatefulSets** also maintain a unique, predictable, enumerated name which in our case will be: `messaging-0`, `messaging-1`, `messaging-2`, etc. Lastly, **StatefulSets** bring up their pods one-at-a-time, solving some initialization / cluster-building problems we've encountered in the past.

The peer discovery plugin, rabbit_peer_discovery_k8s, uses the Kubernetes API to find other nodes. Kubernetes uses Role Based Access Control (RBAC) by default to grant permissions to use the Kubernetes API. Therefore we will need to configure a **ServiceAccount**, **Role**, and **RoleBinding** to allow the plugin to make the requests it needs.

RabbitMQ requires that the hostnames of all cluster members be fully resolvable via DNS. By setting up a **Service** we can let Kubernetes handle the hostname resolution for us. While this isn't new to us, for RabbitMQ it is important to understand exactly how Kubernetes assigns fully qualified domain names (FQDN). By default, it uses the form `<hostname>.<servicename>.<namespace>.svc.cluster.local`. If you don't specify a namespace, you are working in the `default` namespace, therefore we could expect the FQDN for the first node of our RabbitMQ cluster to be `messaging-0.messaging.default.svc.cluster.local`.

To make things easier we will be using a **ConfigMap** to store our custom configs for RabbitMQ. This allows us to put our config files directly inside our YAML definitions and then mount them as volumes.

8.4. Example

Now we'll look at our actual Kubernetes objects. These can be found in `..//example-final/messaging-k8s.yml`.

8.4.1. RBAC

Let's start by establishing a **ServiceAccount** for our pods and binding it to a **Role** that allows us to GET or LIST endpoints for a **Service**:

example-final/messaging-k8s.yml (excerpted)

```
---
apiVersion: v1
kind: ServiceAccount
metadata:
  name: messaging

---
apiVersion: rbac.authorization.k8s.io/v1beta1
kind: Role
metadata:
  name: rabbitmq-peer-discovery-rbac
rules:
  - apiGroups: [""]
    resources: ["endpoints"]
    verbs: ["get", "list"]

---
apiVersion: rbac.authorization.k8s.io/v1beta1
kind: RoleBinding
metadata:
  name: rabbitmq-peer-discovery-rbac
subjects:
  - kind: ServiceAccount
    name: messaging
roleRef:
  apiGroup: rbac.authorization.k8s.io
  kind: Role
  name: rabbitmq-peer-discovery-rbac
```

The **ServiceAccount** `messaging` will be used in our **StatefulSet** so that when a node is brought up, it can query the Kubernetes API to discover the other nodes. You will see this process in the logs later.

8.4.2. ConfigMap

example-final/messaging-k8s.yml (excerpted)

```
---
apiVersion: v1
kind: ConfigMap
metadata:
  name: rabbitmq-config
data:
  enabled_plugins: |
    [rabbitmq_management,rabbitmq_peer_discovery_k8s].
  rabbitmq.conf: |
    cluster_formation.peer_discovery_backend  = rabbit_peer_discovery_k8s
    cluster_formation.k8s.host = kubernetes.default.svc.cluster.local
    cluster_formation.k8s.address_type = hostname
    cluster_formation.node_cleanup.interval = 30
    cluster_formation.node_cleanup.only_log_warning = true
    cluster_partition_handling = autoheal
    queue_master_locator=min-masters
    loopback_users.guest = false
```

The keys and values in the `data` section of a **ConfigMap** are used to hold information that is later placed in a file in a pod template. **ConfigMaps** are mounted as volumes in the template as we will see in a moment.

8.4.3. Services

We will use a **Service** for two purposes:

1. To keep track of what nodes are in the RabbitMQ cluster. The rabbit_peer_discovery_k8s plugin will use this when RabbitMQ is started on a pod.
2. To load balance requests. We can send AMQP traffic *and* HTTP traffic to any node for messaging and administrative interface purposes respectively.

example-final/messaging-k8s.yml (excerpted)

```
---
apiVersion: v1
kind: Service
metadata:
  name: messaging
  labels:
    app: messaging
spec:
  selector:
    app: messaging
  ports:
    - name: amqp
      protocol: TCP
      port: 5672
    - name: http
      protocol: TCP
      port: 15672
```

This is a standard Kubernetes service that will be given a `ClusterIP` and will load balance requests for port `5672` and `15672` (AMQP and RabbitMQ admin interface, respectively).

8.4.4. StatefulSet

example-final/messaging-k8s.yml (excerpted)

```
---
apiVersion: apps/v1
kind: StatefulSet
metadata:
  name: messaging
  labels:
    app: messaging
spec:
  serviceName: messaging
  replicas: 3
  selector:
    matchLabels:
      app: messaging
  template:
    metadata:
      labels:
        app: messaging
    spec:
      serviceAccountName: messaging
      containers:
        - name: rabbitmq
          image: rabbitmq:3.8.3-management
          env:
```

```
          - name: MY_POD_NAME
            valueFrom:
              fieldRef:
                apiVersion: v1
                fieldPath: metadata.name
          - name: RABBITMQ_USE_LONGNAME
            value: "true"
          - name: K8S_SERVICE_NAME
            value: messaging
          - name: K8S_HOSTNAME_SUFFIX
            value: .messaging.default.svc.cluster.local
          - name: RABBITMQ_NODENAME
            value: rabbit@$(MY_POD_NAME).messaging.default.svc.cluster.local
          - name: RABBITMQ_ERLANG_COOKIE
            value: "changeme"
        volumeMounts:
          - name: config-volume
            mountPath: /etc/rabbitmq
      volumes:
        - name: config-volume
          configMap:
            name: rabbitmq-config
            items:
              - key: rabbitmq.conf
                path: rabbitmq.conf
              - key: enabled_plugins
                path: enabled_plugins
```

This should look pretty similar to the **Deployment** we worked on earlier. It creates three replicas by default. Some new things that it has introduced:

- Environment variables can be pulled from Kubernetes parameters, see `MY_POD_NAME` for an example.
- **ConfigMaps** can be mounted in a directory. The keys in the data section serve as file names and the values service as the file contents. You may want to brush up on your YAML multiline strings.
- The environment variables `K8S_SERVICE_NAME` and `K8S_HOSTNAME_SUFFIX` are used by the discovery plugin. If they are not defined it *will* fail.
- `serviceAccountName` is set to messaging to take advantage of our RBAC configuration.

8.4.5. Running the Example

Let's apply our system to a Kubernetes cluster and perform some analysis:

```
PS \example-final> kubectl apply -f .\messaging-k8s.yml
serviceaccount/messaging created
role.rbac.authorization.k8s.io/rabbitmq-peer-discovery-rbac created
rolebinding.rbac.authorization.k8s.io/rabbitmq-peer-discovery-rbac created
configmap/rabbitmq-config created
service/messaging created
statefulset.apps/messaging created
PS C:\Users\rxt1077\it490\example-final> kubectl get pod
NAME          READY   STATUS    RESTARTS   AGE
messaging-0   1/1     Running   0          4m2s
messaging-1   1/1     Running   0          4m1s
messaging-2   1/1     Running   0          4m
```

As you can see, it brings up three pods. Unlike a **Deployment** which uses hashes, the pod names are enumerated. Also unlink a **Deployment** they are brought up one-at-a-time.

Let's look at the logs and see how startup proceeded for `messaging-0`:

```
PS C:\Users\rxt1077\it490\example-final> kubectl logs messaging-0
2020-04-11 18:38:08.118 [info] <0.9.0> Feature flags: list of feature flags found:
2020-04-11 18:38:08.118 [info] <0.9.0> Feature flags:   [ ] drop_unroutable_metric
<snip>
 cookie hash    : TLnIqASP0CKUR3/LGkEZGg==①
<snip>
2020-04-11 18:38:08.301 [info] <0.278.0> Node database directory at
/var/lib/rabbitmq/mnesia/rabbit@messaging-0.messaging.default.
svc.cluster.local is empty. Assuming we need to join an existing cluster or initialise
from scratch...
2020-04-11 18:38:08.301 [info] <0.278.0> Configured peer discovery backend:
rabbit_peer_discovery_k8s
2020-04-11 18:38:08.301 [info] <0.278.0> Will try to lock with peer discovery backend
rabbit_peer_discovery_k8s
2020-04-11 18:38:08.301 [info] <0.278.0> Peer discovery backend does not support
locking, falling back to randomized delay
2020-04-11 18:38:08.301 [info] <0.278.0> Peer discovery backend
rabbit_peer_discovery_k8s supports registration.
2020-04-11 18:38:08.302 [info] <0.278.0> Will wait for 1638 milliseconds before
proceeding with registration...②
2020-04-11 18:38:09.975 [info] <0.278.0> All discovered existing cluster peers:
rabbit@messaging-2.messaging.default.svc.cluster.l
ocal, rabbit@messaging-1.messaging.default.svc.cluster.local, rabbit@messaging-
0.messaging.default.svc.cluster.local
2020-04-11 18:38:09.975 [info] <0.278.0> Peer nodes we can cluster with:
rabbit@messaging-2.messaging.default.svc.cluster.local, r
abbit@messaging-1.messaging.default.svc.cluster.local③
2020-04-11 18:38:09.981 [warning] <0.278.0> Could not auto-cluster with node
rabbit@messaging-2.messaging.default.svc.cluster.loca
l: {error,mnesia_not_running}
2020-04-11 18:38:09.985 [warning] <0.278.0> Could not auto-cluster with node
rabbit@messaging-1.messaging.default.svc.cluster.loca
l: {error,tables_not_present}
2020-04-11 18:38:09.985 [warning] <0.278.0> Could not successfully contact any node
of: rabbit@messaging-2.messaging.default.svc.c
luster.local,rabbit@messaging-1.messaging.default.svc.cluster.local (as in Erlang
distribution). Starting as a blank standalone no
de...④
<snip>
2020-04-11 18:38:11.041 [info] <0.9.0> Server startup complete; 5 plugins started.
 * rabbitmq_management
 * rabbitmq_web_dispatch
 * rabbitmq_management_agent
 * rabbitmq_peer_discovery_k8s
 * rabbitmq_peer_discovery_common
 completed with 5 plugins.
2020-04-11 18:38:11.650 [info] <0.535.0> rabbit on node 'rabbit@messaging-
2.messaging.default.svc.cluster.local' up ⑤
2020-04-11 18:38:11.859 [info] <0.535.0> rabbit on node 'rabbit@messaging-
1.messaging.default.svc.cluster.local' up
```

① This should match the cookie on the other nodes.

② This randomized wait could be optimized since we know we will start in order. See the official example for a better implementation.

③ Other peers were detected, but RabbitMQ was not fully initialized on them.

④ Therefore `messaging-0` became a standalone node.

⑤ Eventually the other nodes joined us.

Let's look at the logs and see how startup proceeded for `messaging-1`:

```
PS example-final> kubectl logs messaging-1
2020-04-11 18:38:09.658 [info] <0.9.0> Feature flags: list of feature flags found:
2020-04-11 18:38:09.658 [info] <0.9.0> Feature flags:   [ ] drop_unroutable_metric
2020-04-11 18:38:09.658 [info] <0.9.0> Feature flags:   [ ] empty_basic_get_metric
<snip>
 cookie hash    : TLnIqASP0CKUR3/LGkEZGg==①
<snip>
2020-04-11 18:38:09.790 [info] <0.278.0> Node database directory at
/var/lib/rabbitmq/mnesia/rabbit@messaging-1.messaging.default.
svc.cluster.local is empty. Assuming we need to join an existing cluster or initialise
from scratch...
2020-04-11 18:38:09.790 [info] <0.278.0> Configured peer discovery backend:
rabbit_peer_discovery_k8s
2020-04-11 18:38:09.791 [info] <0.278.0> Will try to lock with peer discovery backend
rabbit_peer_discovery_k8s
2020-04-11 18:38:09.791 [info] <0.278.0> Peer discovery backend does not support
locking, falling back to randomized delay
2020-04-11 18:38:09.791 [info] <0.278.0> Peer discovery backend
rabbit_peer_discovery_k8s supports registration.
2020-04-11 18:38:09.791 [info] <0.278.0> Will wait for 855 milliseconds before
proceeding with registration...
2020-04-11 18:38:10.670 [info] <0.278.0> All discovered existing cluster peers:
rabbit@messaging-2.messaging.default.svc.cluster.l
ocal, rabbit@messaging-1.messaging.default.svc.cluster.local, rabbit@messaging-
0.messaging.default.svc.cluster.local
2020-04-11 18:38:10.670 [info] <0.278.0> Peer nodes we can cluster with:
rabbit@messaging-2.messaging.default.svc.cluster.local, r
abbit@messaging-0.messaging.default.svc.cluster.local
2020-04-11 18:38:10.673 [warning] <0.278.0> Could not auto-cluster with node
rabbit@messaging-2.messaging.default.svc.cluster.loca
l: {error,tables_not_present}
2020-04-11 18:38:10.696 [info] <0.278.0> Node 'rabbit@messaging-
0.messaging.default.svc.cluster.local' selected for auto-clusterin
g②
<snip>
2020-04-11 18:38:12.118 [info] <0.9.0> Server startup complete; 5 plugins started.
 * rabbitmq_management
 * rabbitmq_web_dispatch
 * rabbitmq_management_agent
 * rabbitmq_peer_discovery_k8s
 * rabbitmq_peer_discovery_common
 completed with 5 plugins.
```

① Sure enough, our cookie is the same

② `messaging-0` is up an available for peering, but `messaging-1` is not. We peered with `messaging-0`

Finally, let's look at the logs and see how startup proceeded for `messaging-2`:

```
PS example-final> kubectl logs messaging-2
2020-04-11 18:38:10.155 [info] <0.9.0> Feature flags: list of feature flags found:
2020-04-11 18:38:10.155 [info] <0.9.0> Feature flags:   [ ] drop_unroutable_metric
<snip>
 cookie hash    : TLnIqASP0CKUR3/LGkEZGg==①
<snip>
2020-04-11 18:38:10.279 [info] <0.287.0> Configured peer discovery backend:
rabbit_peer_discovery_k8s
2020-04-11 18:38:10.279 [info] <0.287.0> Will try to lock with peer discovery backend
rabbit_peer_discovery_k8s
2020-04-11 18:38:10.279 [info] <0.287.0> Peer discovery backend does not support
locking, falling back to randomized delay
2020-04-11 18:38:10.279 [info] <0.287.0> Peer discovery backend
rabbit_peer_discovery_k8s supports registration.
2020-04-11 18:38:10.279 [info] <0.287.0> Will wait for 598 milliseconds before
proceeding with registration...
2020-04-11 18:38:10.891 [info] <0.287.0> All discovered existing cluster peers:
rabbit@messaging-2.messaging.default.svc.cluster.l
ocal, rabbit@messaging-1.messaging.default.svc.cluster.local, rabbit@messaging-
0.messaging.default.svc.cluster.local
2020-04-11 18:38:10.891 [info] <0.287.0> Peer nodes we can cluster with:
rabbit@messaging-1.messaging.default.svc.cluster.local, r
abbit@messaging-0.messaging.default.svc.cluster.local②
2020-04-11 18:38:10.946 [info] <0.287.0> Node 'rabbit@messaging-
1.messaging.default.svc.cluster.local' selected for auto-clusterin
g
<snip>
2020-04-11 18:38:12.009 [info] <0.9.0> Server startup complete; 5 plugins started.
 * rabbitmq_management
 * rabbitmq_web_dispatch
 * rabbitmq_management_agent
 * rabbitmq_peer_discovery_k8s
 * rabbitmq_peer_discovery_common
```

① Same cookie as all the other nodes, good.

② `messaging-2` could peer with either `messaging-0` or `messaging-1` as it was started last. It chose `messaging-1`.

The last thing we can do is look at the output of `rabbitmqctl cluster_status` to see how our cluster is running. Executing this command on any node will tell you about the health of the entire RabbitMQ cluster:

```
PS example-final> kubectl exec -it messaging-0 -- rabbitmqctl cluster_status
Cluster status of node rabbit@messaging-0.messaging.default.svc.cluster.local ...
Basics

Cluster name: rabbit@messaging-0.messaging.default.svc.cluster.local

Disk Nodes
```

```
rabbit@messaging-0.messaging.default.svc.cluster.local
rabbit@messaging-1.messaging.default.svc.cluster.local
rabbit@messaging-2.messaging.default.svc.cluster.local

Running Nodes

rabbit@messaging-0.messaging.default.svc.cluster.local
rabbit@messaging-1.messaging.default.svc.cluster.local
rabbit@messaging-2.messaging.default.svc.cluster.local

Versions

rabbit@messaging-0.messaging.default.svc.cluster.local: RabbitMQ 3.8.3 on Erlang
22.3.1
rabbit@messaging-1.messaging.default.svc.cluster.local: RabbitMQ 3.8.3 on Erlang
22.3.1
rabbit@messaging-2.messaging.default.svc.cluster.local: RabbitMQ 3.8.3 on Erlang
22.3.1

Alarms

(none)

Network Partitions

(none)

Listeners

Node: rabbit@messaging-0.messaging.default.svc.cluster.local, interface: [::], port:
25672, protocol: clustering, purpose: inter-n
ode and CLI tool communication
Node: rabbit@messaging-0.messaging.default.svc.cluster.local, interface: [::], port:
5672, protocol: amqp, purpose: AMQP 0-9-1 and
 AMQP 1.0
Node: rabbit@messaging-0.messaging.default.svc.cluster.local, interface: [::], port:
15672, protocol: http, purpose: HTTP API
Node: rabbit@messaging-1.messaging.default.svc.cluster.local, interface: [::], port:
25672, protocol: clustering, purpose: inter-n
ode and CLI tool communication
Node: rabbit@messaging-1.messaging.default.svc.cluster.local, interface: [::], port:
5672, protocol: amqp, purpose: AMQP 0-9-1 and
 AMQP 1.0
Node: rabbit@messaging-1.messaging.default.svc.cluster.local, interface: [::], port:
15672, protocol: http, purpose: HTTP API
Node: rabbit@messaging-2.messaging.default.svc.cluster.local, interface: [::], port:
25672, protocol: clustering, purpose: inter-n
ode and CLI tool communication
Node: rabbit@messaging-2.messaging.default.svc.cluster.local, interface: [::], port:
5672, protocol: amqp, purpose: AMQP 0-9-1 and
```

```
 AMQP 1.0
Node: rabbit@messaging-2.messaging.default.svc.cluster.local, interface: [::], port:
15672, protocol: http, purpose: HTTP API

Feature flags

Flag: drop_unroutable_metric, state: enabled
Flag: empty_basic_get_metric, state: enabled
Flag: implicit_default_bindings, state: enabled
Flag: quorum_queue, state: enabled
Flag: virtual_host_metadata, state: enabled
```

This shows us that there are three nodes, the nodes are all running the same version of RabbitMQ and Erlang, and that they are listening for AMQP and admin interface connections.

8.5. Questions

1. *What sets RabbitMQ clustering apart from more traditional primary / standby replication?*

2. *What is the difference between a **Deployment** and a **StatefulSet**? Why did we choose a **StatefulSet** for this application?*

3. *Why does our peer discovery plugin use the Kubernetes API and what alternatives are there?*

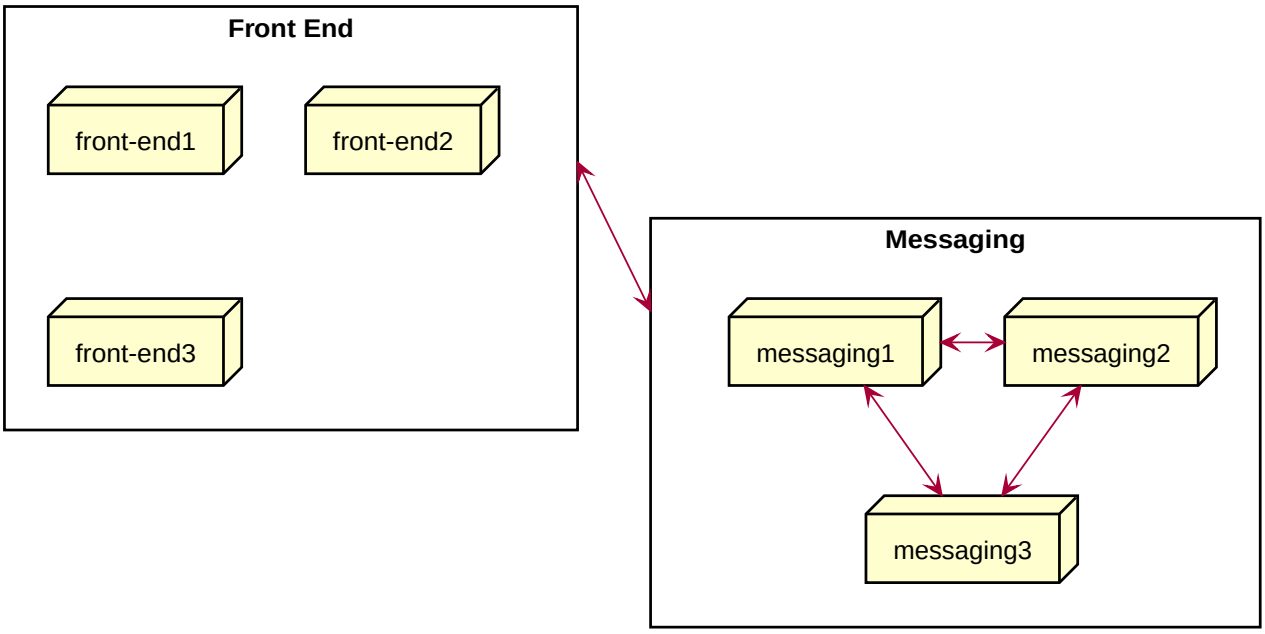
4. *What role does RBAC play in the Kubernetes cluster?*

5. *What does a **ConfigMap** do and how is it used?*

Chapter 9. Front End in Kubernetes

9.1. Introduction

Migrating **Front End** to Kubernetes should be a relatively simple. It is already designed with horizontal scaling in mind. All communication with the other components is handled by **Messaging** and **Front End** does not maintain any state. Multiple **Front Ends** can be run at the same time and as far as the client is concerned, any instance can be used. Kubernetes **Services** can be used to manage our **Messaging** and **Front End** instances, making connections easy:



Notice how the front-end instances don't need to communicate with each other unlike the messaging instances which are part of a cluster. This makes scaling much less complex.

9.2. Kubernetes

Front End requires a way of accessing a **Service** from the outside world. The traditional way of doing this is through a load balancer, which has an external IP and forwards traffic from a standard port, 80 for HTTP or 443 for HTTPS, to the **Service** and ultimately one of the running pods. The concept of load balancing isn't new to us, we have been using it implicitly when we create a **Service**. The new concept is a method of external access.

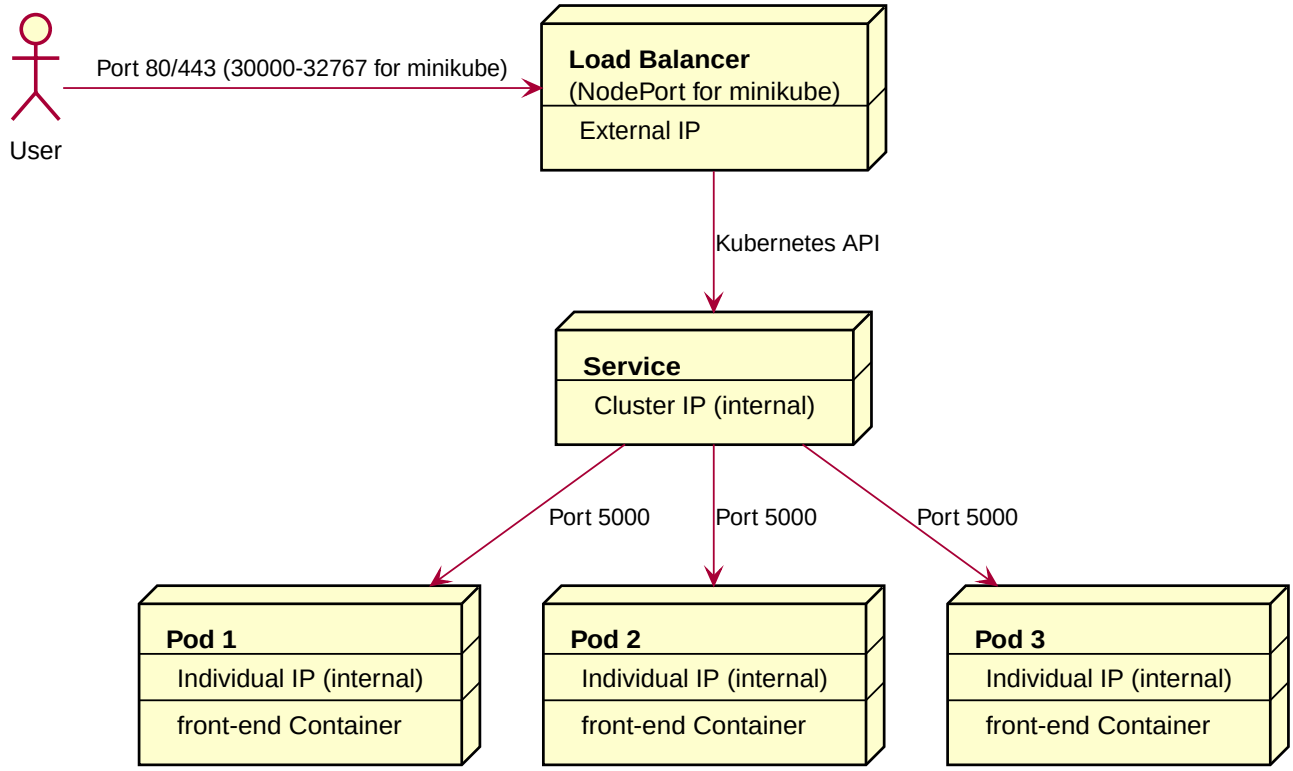


Figure 8. Load Balancer Architecture

Our Flask application will listen on port 5000 by default. In an actual production environment (not minikube) the load balancer would be given an external IP listening on a standard port. Minikube will support the definition of a LoadBalancer type **Service** object, but it will actually use a slightly simpler object called a NodePort to access the service. For us, this means we can get to our service via a local IP and a random port between 30000 and 32767. The `minikube service` command will automatically open the URL for the service in your default web browser.

9.3. Example

9.3.1. Updating the Docker Image

The Docker image for use in Kubernetes can actually be simplified from what we were running before. We can remove the `wait-for-it.sh` script and we no longer have to call it from the Dockerfile:

example-final/front-end/Dockerfile

```
FROM python:3.9.0a5-buster
COPY . /app
WORKDIR /app
RUN pip install -r requirements.txt
ENV FLASK_APP=app.py
CMD ["flask", "run", "--host=0.0.0.0"]
```

Our **Service** name for **Messaging** is still messaging, so we don't even need to change the hostname the Messaging class connects to. If you look at the source code you will see that I just changed the comment to reference Kubernetes instead of Docker Compose.

9.3.2. Building the Docker Image

In order for Kubernetes to be able to use our custom image, we need to build it and make it available to the Docker daemon running *inside* minikube. With minikube started, but without the environment set up correctly, `docker ps` will only show the containers you have running natively on the host:

```
PS example-final> docker ps
CONTAINER ID  IMAGE                                  COMMAND
9689f05c2fca  gcr.io/k8s-minikube/kicbase:v0.0.8   "/usr/local/bin/entr⋯"
```

In this example the only container I have running is the container used by the minikube `docker` driver. If you are running it under `virtualbox` or `hyperv` you may not see any containers running. If you don't have docker running on your host, you may not even be able to execute the `docker ps` command.

Now if we execute the `minikube docker-env` command and follow the directions, `docker ps` should show the entire Kubernetes environment running in containers as it is querying the Docker daemon running *inside* minikube:

```
PS example-final> minikube docker-env ①
$Env:DOCKER_TLS_VERIFY = "1"
$Env:DOCKER_HOST = "tcp://127.0.0.1:32769"
$Env:DOCKER_CERT_PATH = "C:\Users\rxt1077\.minikube\certs"
$Env:MINIKUBE_ACTIVE_DOCKERD = "minikube"
# To point your shell to minikube's docker-daemon, run:
# & minikube -p minikube docker-env | Invoke-Expression ②
PS example-final> minikube docker-env | Invoke-Expression ③
PS example-final> docker ps ④
CONTAINER ID        IMAGE                  COMMAND                  CREATED
47eff1ee88b2        67da37a9a360           "/coredns -conf /etc…"   15 hours ago
79d53aceb8f2        67da37a9a360           "/coredns -conf /etc…"   15 hours ago
c0eebfebded2        aa67fec7d7ef           "/bin/kindnetd"          15 hours ago
b6a95759fdbe        43940c34f24f           "/usr/local/bin/kube…"   15 hours ago
2173eeb16643        k8s.gcr.io/pause:3.2   "/pause"                 15 hours ago
3b72a0aa725b        4689081edb10           "/storage-provisioner"   15 hours ago
4616fd610301        k8s.gcr.io/pause:3.2   "/pause"                 15 hours ago
78e245e291fb        k8s.gcr.io/pause:3.2   "/pause"                 15 hours ago
764d41d70f58        k8s.gcr.io/pause:3.2   "/pause"                 15 hours ago
29f46b453297        k8s.gcr.io/pause:3.2   "/pause"                 15 hours ago
fa87bf3bdcfb        a31f78c7c8ce           "kube-scheduler --au…"   15 hours ago
5e51df5cc257        d3e55153f52f           "kube-controller-man…"   15 hours ago
dc051639dbed        74060cea7f70           "kube-apiserver --ad…"   15 hours ago
01cb4068fc8b        303ce5db0e90           "etcd --advertise-cl…"   15 hours ago
efb620d9f59a        k8s.gcr.io/pause:3.2   "/pause"                 15 hours ago
f723dce3ac9d        k8s.gcr.io/pause:3.2   "/pause"                 15 hours ago
89d58b537cae        k8s.gcr.io/pause:3.2   "/pause"                 15 hours ago
d05cf6dbe82a        k8s.gcr.io/pause:3.2   "/pause"                 15 hours ago
```

① Running docker-env by itself prints out how the command should be executed

② Here it is telling us how to run it

③ Now we actually run it as recommended and change the environment

④ `docker ps` now lists everything running on the minikube docker daemon

Now we can build our front-end image and it will be available to minikube:

```
PS example-final> docker build -t front-end:v1 ./front-end
Sending build context to Docker daemon  9.728kB
Step 1/6 : FROM python
latest: Pulling from library/python
7e2b2a5af8f6: Pull complete
09b6f03ffac4: Pull complete
dc3f0c679f0f: Pull complete
fd4b47407fc3: Pull complete
b32f6bf7d96d: Pull complete
3940e1b57073: Pull complete
ce1fce2a6cf9: Pull complete
1f593157bb4c: Pull complete
```

```
bde1ccd8f1b8: Pull complete
Digest: sha256:3df040cc8e804b731a9e98c82e2bc5cf3c979d78288c28df4f54bbdc18dbb521
Status: Downloaded newer image for python:latest
 ---> b55669b4130e
Step 2/6 : COPY . /app
 ---> b88600cc635a
Step 3/6 : WORKDIR /app
 ---> Running in 20bc72069ed8
Removing intermediate container 20bc72069ed8
 ---> 61eb3608a02a
Step 4/6 : RUN pip install -r requirements.txt
 ---> Running in da9520ffee48
Collecting Flask
  Downloading Flask-1.1.2-py2.py3-none-any.whl (94 kB)
Collecting pika
  Downloading pika-1.1.0-py2.py3-none-any.whl (148 kB)
Collecting itsdangerous>=0.24
  Downloading itsdangerous-1.1.0-py2.py3-none-any.whl (16 kB)
Collecting Werkzeug>=0.15
  Downloading Werkzeug-1.0.1-py2.py3-none-any.whl (298 kB)
Collecting click>=5.1
  Downloading click-7.1.1-py2.py3-none-any.whl (82 kB)
Collecting Jinja2>=2.10.1
  Downloading Jinja2-2.11.2-py2.py3-none-any.whl (125 kB)
Collecting MarkupSafe>=0.23
  Downloading MarkupSafe-1.1.1-cp38-cp38-manylinux1_x86_64.whl (32 kB)
Installing collected packages: itsdangerous, Werkzeug, click, MarkupSafe, Jinja2,
Flask, pika
Successfully installed Flask-1.1.2 Jinja2-2.11.2 MarkupSafe-1.1.1 Werkzeug-1.0.1
click-7.1.1 itsdangerous-1.1.0 pika-1.1.0
Removing intermediate container da9520ffee48
 ---> 8d2f4da8b8b4
Step 5/6 : ENV FLASK_APP=app.py
 ---> Running in 4cdf7ad5a96e
Removing intermediate container 4cdf7ad5a96e
 ---> 4b5853571124
Step 6/6 : CMD ["flask", "run", "--host=0.0.0.0"]
 ---> Running in ff512bc5e42b
Removing intermediate container ff512bc5e42b
 ---> 52ec5d015433
Successfully built 52ec5d015433
Successfully tagged front-end:v1
```

Make sure you give your image a tag with a version ("v1" in our example). Kubernetes will automatically try to pull the "latest" version for untagged images and since we are not using a Docker image repository that pull will fail.

9.3.3. Service

Let's take a look at our **Service** definition:

example-final/front-end-k8s.yml (excerpted)

```
---
apiVersion: v1
kind: Service
metadata:
  name: front-end
  labels:
    app: front-end
spec:
  type: LoadBalancer
  selector:
    app: front-end
  ports:
    - name: http
      protocol: TCP
      port: 80
      targetPort: 5000
```

The only new things in this definition are `type: LoadBalancer` in the `spec` and `targetPort` in the `ports` list. This allows us to access this service externally on port 80 and have it routed internally to port 5000 on one of the pods. It is worth noting that in a real-life scenario, this will create a load balancer with an external IP via your IaaS provider. These cost money and it can add up as you expose more services to the outside world. Fortunately, as explained in the previous Kubernetes section, we can still use minikube to test externally connecting to our service with the `minikube service` command.

9.3.4. Deployment

For **Front End** we can use a simple deployment:

example-final/front-end-k8s.yml (excerpted)

```
---
apiVersion: apps/v1
kind: Deployment
metadata:
  name: front-end
  labels:
    app: front-end
spec:
  replicas: 3
  selector:
    matchLabels:
      app: front-end
  template:
    metadata:
      labels:
        app: front-end
    spec:
      containers:
        - name: front-end
          image: front-end:v1
          env:
            - name: RABBITMQ_DEFAULT_USER
              value: "guest"
            - name: RABBITMQ_DEFAULT_PASS
              value: "guest"
            - name: FLASK_SECRET_KEY
              value: "changeme"
```

9.3.5. Running the Example

Now let's apply both **Messaging** and **Front End**. This will let us check to see if the **Front End** serves web pages *and* if the **Front End** can connect to **Messaging** and create queues. We won't be able to login / register users entirely yet because we don't have **Back End** running.

```
PS example-final> kubectl apply -f messaging-k8s.yml -f front-end-k8s.yml ①
serviceaccount/messaging created
role.rbac.authorization.k8s.io/rabbitmq-peer-discovery-rbac created
rolebinding.rbac.authorization.k8s.io/rabbitmq-peer-discovery-rbac created
configmap/rabbitmq-config created
service/messaging created
statefulset.apps/messaging created
service/front-end created
deployment.apps/front-end created
PS example-final> kubectl get pod ②
NAME                         READY   STATUS    RESTARTS   AGE
front-end-7f7c4f5455-6qbcx   1/1     Running   0          4h46m
front-end-7f7c4f5455-htdlv   1/1     Running   0          4h46m
front-end-7f7c4f5455-q2nn6   1/1     Running   0          4h46m
messaging-0                  1/1     Running   0          16m
messaging-1                  1/1     Running   0          16m
messaging-2                  1/1     Running   0          16m
PS example-final> minikube service front-end ③
|-----------|-----------|-------------|---------------------------|
| NAMESPACE |   NAME    | TARGET PORT |            URL            |
|-----------|-----------|-------------|---------------------------|
| default   | front-end | http/5000   | http://192.168.135.5:31232 |
|-----------|-----------|-------------|---------------------------|
* Opening service default/front-end in default browser...
```

① The `kubectl apply` command can be used with multiple files or all YAML files in a directory. Here we specify the two components we want to start.

② After a little while, you should see six pods running: three for **Messaging** and three for **Front End**.

③ This command will start the default browser and pass it the URL to the front-end **Service**. If you just want the URL instead of having it open the browser you can use `kubectl service --url front-end`.

If you apply the objects, run the `kubectl get pod` command, and see `ErrImagePull` or `ImagePullBackOff` it means that Kubernetes can't pull your Docker images. Either you've misnamed a stock image that in the `name` attribute of your container, or you are trying to use a custom image that you did not make available to minikube. See Building the Docker Image.

If your connections are timing out with the `minikube service` command and you are using the minikube `docker` driver, try with `hyperv` or `virtualbox`. The `docker` driver seems to have some issues with port forwarding.

We see our **Front End** website in our default browser. If we try to register a user we get a message saying "No response from back end."

If you want to test things quickly, without opening a browser the curl command is a great thing to know. In PowerShell curl is an alias to Invoke-WebRequest, but it will still work for simple testing. Try `curl $(minikube service --url front-end)`.

Now let's run a shell in our RabbitMQ cluster and use the `rabbitmqctl` command to verify that a `requests` queue has actually been created:

```
PS C:\Users\rxt1077\it490\example-final> kubectl exec -it messaging-0 -- bash ①
root@messaging-0:/# rabbitmqctl list_queues
Timeout: 60.0 seconds ...
Listing queues for vhost / ...
name    messages
request 1 ②
root@messaging-0:/# exit
exit
```

① It doesn't matter which node we run a shell on, they should all be able to see all queues. I chose `messaging-0` because it is easy to remember.

② There is a request queue, with one message in it.

9.4. Questions

1. *Why is it easier to set up **Front End** in Kubernetes than it is to set up **Messaging**?*

2. *What does a LoadBalancer type **Service** do?*

3. *When creating custom images for use with minikube, why do you have to set up your Docker environment variables before building images?*

4. *How do you make a **Service** accessible from outside the Kubernetes cluster?*

5. *If you wanted to use the RabbitMQ web-based admin interface for testing, what would you have to do to access it?*

Chapter 10. Back End in Kubernetes

The last key to our Kubernetes migration is a functioning **Back End**. In this section we will build the Kubernetes object (yes, singular) needed to support this role.

10.1. Introduction

Back End is our conduit between **Messaging** and **Database** and we have implemented it as a Python script. Replicas of **Back End** can function independently since messages can only be pulled off the queue one-at-a-time, they are removed after they are pulled, and an exclusive response queue is created for the response:

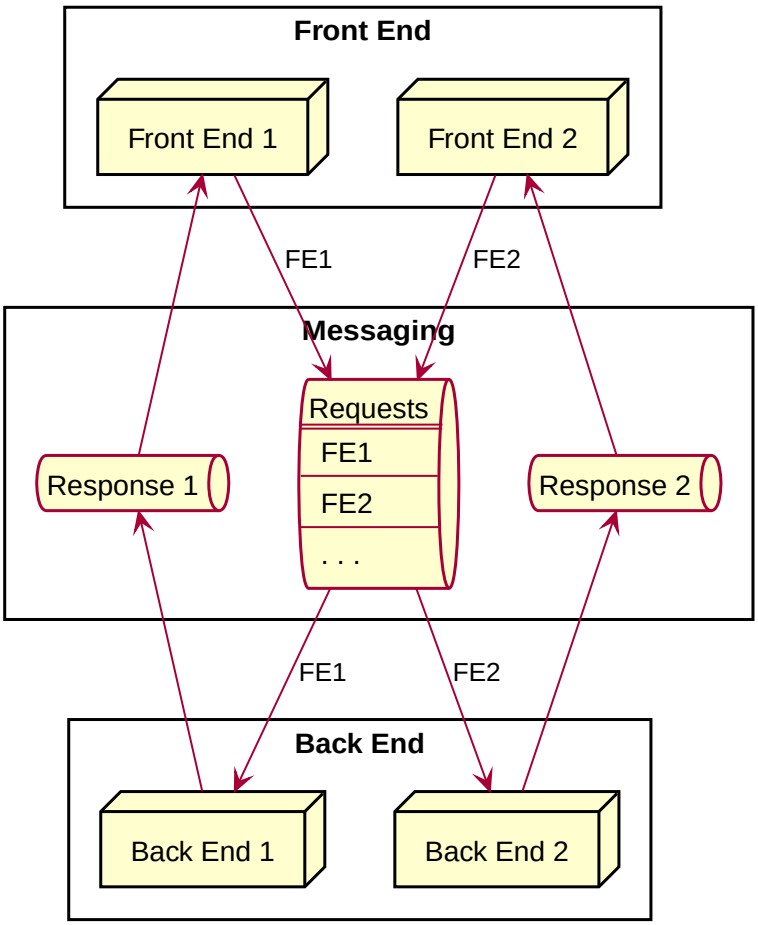


Figure 9. **Back End** *Architecture*

In the above diagram two different **Front Ends** are communicating with **Messaging**. Two different **Back Ends** are pulling requests (FE1, FE2, etc.) out of the requests queue and processing them. Notice that each **Front End** has a separate response queue (that can be derived from the message in the requests queue). This architecture allows multiple **Backends** to run at the same time without needing to be in communication with each other.

We do not need to learn about any new Kubernetes objects to migrate **Back End**. It does not require an external connection to any port or even an internal connection to a port, therefore a **Service** object isn't even required.

10.2. Example

The only changes you will see in **Back Ends** application code is separating database requests into read and read / write, to correspond to our new load-balanced service. Here is the new connect sequence:

example-final/back-end/app.py (excerpted)

```
        logging.info("Connecting to the read-only database...")
        postgres_password = os.environ['POSTGRES_PASSWORD']
        conn_r = psycopg2.connect(
            host='db-r',
            database='example',
            user='postgres',
            password=postgres_password
        )

        logging.info("Connecting to the read-write database...")
        postgres_password = os.environ['POSTGRES_PASSWORD']
        conn_rw = psycopg2.connect(
            host='db-rw',
            database='example',
            user='postgres',
            password=postgres_password
        )
```

This makes `curr_r`, `conn_r`, `curr_rw`, `conn_rw` available for read and read / write requests respectively. `process_request` then uses correct connection depending on the action:

example-final/back-end/app.py (excerpted)

```
def process_request(ch, method, properties, body):
    """
    Gets a request from the queue, acts on it, and returns a response to the
    reply-to queue
    """
    request = json.loads(body)
    if 'action' not in request:
        response = {
            'success': False,
            'message': "Request does not have action"
        }
    else:
        action = request['action']
        if action == 'GETHASH':
            data = request['data']
            email = data['email']
            logging.info(f"GETHASH request for {email} received")
            curr_r.execute('SELECT hash FROM users WHERE email=%s;', (email,))
            row =  curr_r.fetchone()
            if row == None:
                response = {'success': False}
            else:
                response = {'success': True, 'hash': row[0]}
        elif action == 'REGISTER':
            data = request['data']
            email = data['email']
            hashed = data['hash']
            logging.info(f"REGISTER request for {email} received")
            curr_r.execute('SELECT * FROM users WHERE email=%s;', (email,))
            if curr_r.fetchone() != None:
                response = {'success': False, 'message': 'User already exists'}
            else:
                curr_rw.execute('INSERT INTO users VALUES (%s, %s);', (email, hashed))
                conn_rw.commit()
                response = {'success': True}
        else:
            response = {'success': False, 'message': "Unknown action"}
    logging.info(response)
    ch.basic_publish(
        exchange='',
        routing_key=properties.reply_to,
        body=json.dumps(response)
    )
```

The Dockerfile does not require any changes, but minikube does requires that an image be built and available. The only object we need to create is a **Deployment**:

example-final/back-end-k8s.yml

```
---
apiVersion: apps/v1
kind: Deployment
metadata:
  name: back-end
  labels:
    app: back-end
spec:
  replicas: 3
  selector:
    matchLabels:
      app: back-end
  template:
    metadata:
      labels:
        app: back-end
    spec:
      containers:
        - name: back-end
          image: back-end:v1
          env:
            - name: RABBITMQ_DEFAULT_USER
              value: "guest"
            - name: RABBITMQ_DEFAULT_PASS
              value: "guest"
            - name: POSTGRES_PASSWORD
              value: "changeme"
```

Now let's build and tag our back-end:v1 image to make it available to minikube:

```
PS example-final> minikube docker-env | Invoke-Expression ①
PS example-final> cd .\back-end\
PS example-final\back-end> docker build -t back-end:v1 . ②
Sending build context to Docker daemon  7.168kB
Step 1/5 : FROM python
 ---> b55669b4130e
Step 2/5 : COPY . /app
 ---> 6aacbd4f55d8
Step 3/5 : WORKDIR /app
 ---> Running in 5f993e9c691d
Removing intermediate container 5f993e9c691d
 ---> c8b736fd9f9c
Step 4/5 : RUN pip install -r requirements.txt
 ---> Running in f423d983860c
Collecting pika
  Downloading pika-1.1.0-py2.py3-none-any.whl (148 kB)
Collecting psycopg2
  Downloading psycopg2-2.8.5.tar.gz (380 kB)
Building wheels for collected packages: psycopg2
  Building wheel for psycopg2 (setup.py): started
  Building wheel for psycopg2 (setup.py): finished with status 'done'
  Created wheel for psycopg2: filename=psycopg2-2.8.5-cp38-cp38-linux_x86_64.whl
size=500514 sha256=6a53ea80799efeaeb8f4aeec9b7e
b4d7fe0451d9efb02258f9a57801fa5d1b0a
  Stored in directory:
/root/.cache/pip/wheels/35/64/21/9c9e2c1bb9cd6bca3c1b97b955615e37fd309f8e8b0b9fdf1a
Successfully built psycopg2
Installing collected packages: pika, psycopg2
Successfully installed pika-1.1.0 psycopg2-2.8.5
Removing intermediate container f423d983860c
 ---> af068e0b4647
Step 5/5 : CMD ["python", "app.py"]
 ---> Running in 969562a251ec
Removing intermediate container 969562a251ec
 ---> a1f03249f42c
Successfully built a1f03249f42c
Successfully tagged back-end:v1
```

① Don't forget to have your environment set up to build for the minikube docker daemon. I started a new terminal to run this, so I had to set up the environment again.

② Don't forget to use a tag, back-end:v1 in this case.

Now we'll apply our minikube objects for the **Back End**:

```
PS example-final> kubectl apply -f .\back-end-k8s.yml
deployment.apps/back-end created
PS example-final> kubectl get pod
NAME                        READY   STATUS             RESTARTS   AGE
back-end-7685957868-fbzqk   1/1     Running            0          3s
back-end-7685957868-t4n9x   1/1     Running            0          3s
back-end-7685957868-ws2ss   1/1     Running            0          3s
PS example-final> kubectl logs back-end-7685957868-fbzqk
INFO:root:Waiting 1s...
INFO:root:Connecting to the database...
INFO:root:Waiting 2s...
INFO:root:Connecting to the database...
INFO:root:Waiting 4s...
INFO:root:Connecting to the database...
INFO:root:Waiting 8s...
INFO:root:Connecting to the database...
INFO:root:Waiting 16s...
```

As you can see, it is waiting for **Database** to start up. Since we have all of the components, why don't we try bringing the entire system up? `kubectl apply -f .` will apply *all* of the YAML files in the current directory. Since we are using the `apply` command, only changes that are needed to reach the state of the objects in the files will be made. Lastly, we need to make sure that the `front-end:v1` image is built and available:

```
PS example-final> docker build -t front-end:v1 ./front-end
Sending build context to Docker daemon   16.9kB
Step 1/6 : FROM python
latest: Pulling from library/python
90fe46dd8199: Pull complete
35a4f1977689: Pull complete
bbc37f14aded: Pull complete
74e27dc593d4: Pull complete
4352dcff7819: Pull complete
deb569b08de6: Pull complete
98fd06fa8c53: Pull complete
7b9cc4fdefe6: Pull complete
e8e1fd64f499: Pull complete
Digest: sha256:adcfb73e4ca83b126cc3275f3851c73aecca20e59a48782e9ddebb3a88e57f96
Status: Downloaded newer image for python:latest
 ---> a6be143418fc
Step 2/6 : COPY . /app
 ---> d0441d56a485
Step 3/6 : WORKDIR /app
 ---> Running in 31809274a574
Removing intermediate container 31809274a574
 ---> 4cd78efa655a
Step 4/6 : RUN pip install -r requirements.txt
 ---> Running in 7c1603cf2503
Collecting Flask
```

```
  Downloading Flask-1.1.2-py2.py3-none-any.whl (94 kB)
Collecting pika
  Downloading pika-1.1.0-py2.py3-none-any.whl (148 kB)
Collecting Werkzeug>=0.15
  Downloading Werkzeug-1.0.1-py2.py3-none-any.whl (298 kB)
Collecting Jinja2>=2.10.1
  Downloading Jinja2-2.11.2-py2.py3-none-any.whl (125 kB)
Collecting click>=5.1
  Downloading click-7.1.1-py2.py3-none-any.whl (82 kB)
Collecting itsdangerous>=0.24
  Downloading itsdangerous-1.1.0-py2.py3-none-any.whl (16 kB)
Collecting MarkupSafe>=0.23
Installing collected packages: Werkzeug, MarkupSafe, Jinja2, click, itsdangerous,
Flask, pika
Successfully installed Flask-1.1.2 Jinja2-2.11.2 MarkupSafe-1.1.1 Werkzeug-1.0.1
click-7.1.1 itsdangerous-1.1.0 pika-1.1.0
Removing intermediate container 7c1603cf2503
 ---> a6a9f8d6b40c
Step 5/6 : ENV FLASK_APP=app.py
 ---> Running in 008548ce7dfb
Removing intermediate container 008548ce7dfb
 ---> 28fce011bbd3
Step 6/6 : CMD ["flask", "run", "--host=0.0.0.0"]
 ---> Running in 7adb0edc6b4e
Removing intermediate container 7adb0edc6b4e
 ---> 34f3b5c20f75
Successfully built 34f3b5c20f75
Successfully tagged front-end:v1
```

Now we can bring everything in the directory up with the `kubectl apply -f .` command:

```
PS C:\Users\rxt1077\it490\example-final> kubectl apply -f .
deployment.apps/back-end unchanged
persistentvolumeclaim/db-primary-pv-claim created
service/db-rw created
service/db-r created
deployment.apps/db-rw created
deployment.apps/db-r created
service/front-end created
deployment.apps/front-end created
serviceaccount/messaging created
role.rbac.authorization.k8s.io/rabbitmq-peer-discovery-rbac created
rolebinding.rbac.authorization.k8s.io/rabbitmq-peer-discovery-rbac created
configmap/rabbitmq-config created
service/messaging created
statefulset.apps/messaging created
```

Finally, running the `minikube service front-end` command should start the default browser with the URL needed to access **Front End**. You can test registering and logging in as a user.

10.3. Questions

1. *Why did we have to change the application code for **Back End**?*

2. *Why doesn't **Back End** need a **Service**?*

3. *How do we make sure that the `back-end:v1` image is available to minikube?*

4. *What particular issues, with regard to the entire system, might horizontally scaling **Back End** help with?*

5. *How can you apply more than one YAML file with the `kubectl` command?*

Chapter 11. Google Kubernetes Engine

11.1. Introduction

Several times over the course of this text you probably asked yourself, "Why are we doing it this way?" Hopefully the chapters answered most of those questions. This chapter hopes to answer the question, "Why did we migrate to Kubernetes?" The answer being, "To create a scalable system that can be run on enterprise-grade hardware." The best way to see what that looks like is to actually do it.

In this chapter we will deploy our full system on Google Kubernetes Engine. If you want to follow through the examples on your own, you will need to sign up for a Google Cloud login. Please note that while Google Cloud does have a free tier, you will still need to a credit card to your account and *you can be charged money for the services you use*. If you are worried about incuring an expense, feel free to simple read through the examples.

11.2. Installing gcloud

We will be doing as much of this is possible from the command line, but we will need to install a CLI to interact with Google cloud. The `gcloud` command is installed as part of the Google Cloud SDK. Install it using the Google Cloud SDK Interactive Installer and follow the prompts to run `gcloud init` once it is installed (this is the default).

The init procedure will open the default browser and prompt you to sign in with a Google account that you created a Google Cloud login on. Once authenticated, you can refer back to the terminal it opened and `Create a new project`, with any unique name you can think of. The project id *must* be globally unique, so you may want to use the date as part of your ID: `example-2020428`.

11.3. Switching Clusters in Kubectl

11.4. Resources

- Google Kubernetes Engine Quickstart
- Google Cloud SDK Interactive Installer