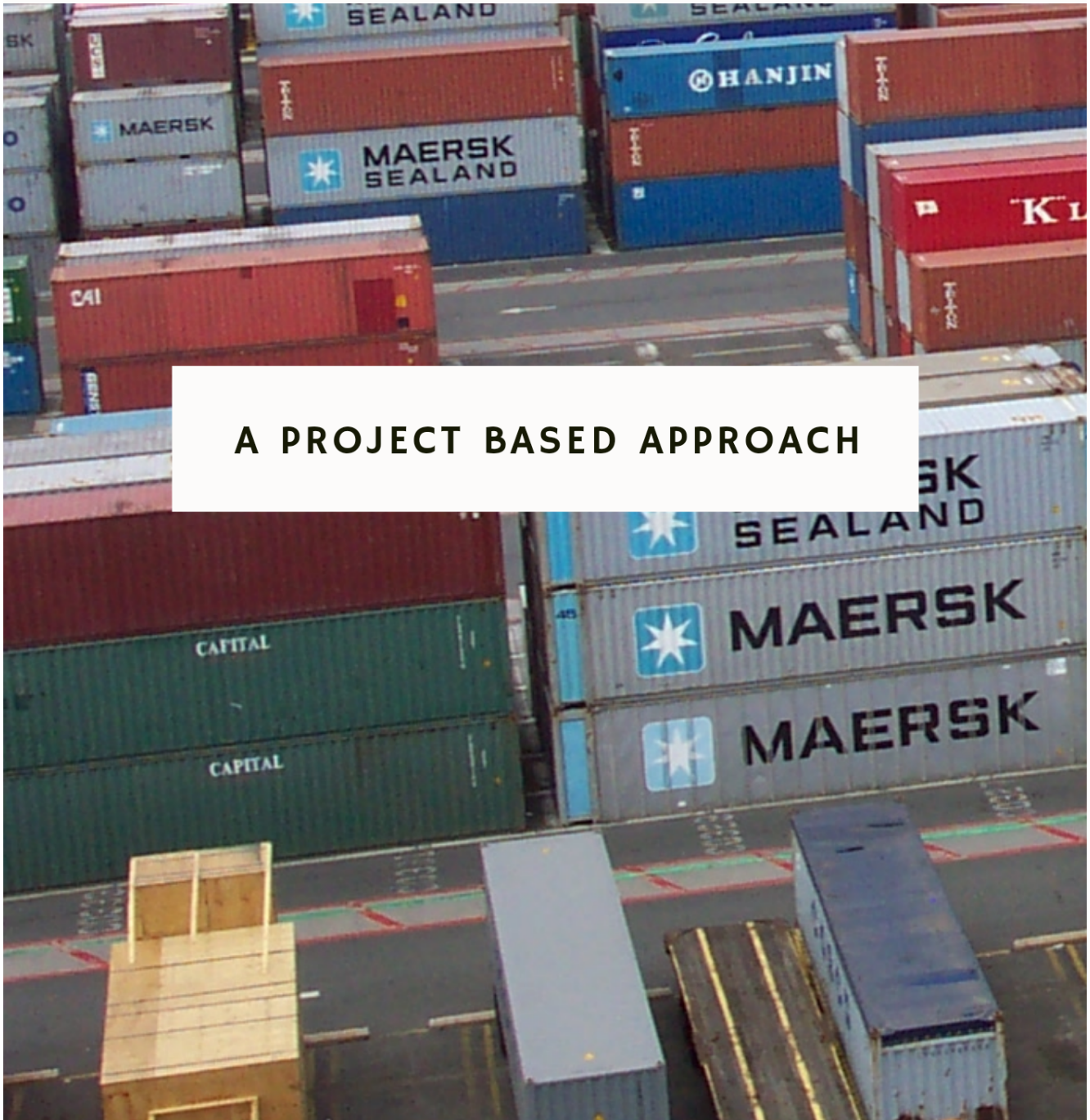
SYSTEMS INTEGRATION



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. Modern tools like Docker and Kubernetes are used to allow the reader to develop the system on their own machine, but still be able to deploy to an enterprise cluster by the end of the text. A computer that meets these minimum specifications will be required to complete the coursework.

The process of systems integration can be thought of as building a jigsaw puzzle. Every puzzle is unique because the pieces and the way they fit together are different. It would be impossible to create a text that outlines step-by-step how to solve every puzzle. In the same way it is impossible to create a text that tells you how to integrate all types of systems. This text aims to provide the basic scaffolding to build a project with ample materials for further research. The readers is expected to use these resources as they tackle the issues within their own project.

In accordance with the show-me-the-code attitude of the open source software movement, all of the code for the text and examples is 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

Acknowledgements

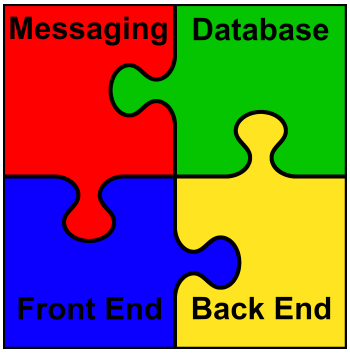
The author would like to acknowledge the hard work of DJ Kehoe, whose devotion to the success of his students led him to create a rigorous, project-based Systems Integration course. Without his work, none of this would be possible.

This book also would not have been possible without Asciidoctor an open source text processor and publishing toolchain. Thank you to Dan Allen, Sarah White, Ryan Waldron, Nick Hengeveld and the Asciidoctor project contributers for making this publishing tool free and open.

PlantUML was used for diagram creation and the author would like to thank Arnaud Roques for creating such a versatile, open source diagram tool.

The cover photo was taken by Captain Albert E. Theberge, NOAA Corps (ret.)

Project



This text is meant to accompany the creation of a group project. The project is designed to be completed by a four person group over a 15 week semester. If you are using this text as part of a course, all parts of this section are subject to change / augmentation by your instructor. If you are using this text as an independent, self-study tool you should still be able to complete all of the milestones and create the deliverables although it may take you longer. What follows is the typical assignment structure for developing the project:

Project Proposal

The project proposal and the comments on it function as a record of dialog between the instructor and the group. You can think of the proposal as a contract showing what is required to achieve a good grade on the project. The instructor may specify additional deliverables depending on what your group has chosen to work on, or they may walk-back your proposal if they believe it is too difficult.

Developing a Proposal

A data-first approach often helps in coming up with a project. Think of a data source that could be used to create a marketable software as a service (SaaS) product. For the purposes of this project, the data source needs to be freely available. Use the following questions to guide your proposal:

- What data will you use?
- What service will you offer with this data?
- Why would people be willing to pay money for your service?
- Who would be willing to pay money for your service?

Here is an example that you can *not* use:

Example 1. Project Proposal

> My group will design a web app that allows people in New Jersey to create itineraries for upcoming events based on the current weather forecast. We will use data from OpenWeatherMaps.org as well NJ.com to plan separate events based on how filled people want their schedule and what the weather is predicted to be. Our target demographic is busy professionals who are comfortable using technology. Since they already use services like Google Calendar for keeping track of their events, why not try using a service that will suggest upcoming events? The time they save planning would easily be worth the cost of the service.

Milestones

Milestones exist to encourage groups to keep pace with the project. This is not the kind of project that can be completed in the last week of class. All milestones should be entered in GitHub and linked with Issues assigned to specific members of the group. Individual contributions of group members will be assessed through GitHub activity so be sure you are actively participating.

One

- The group has four members.
- Each member of the group has read the Git, GitHub, YAML, and Docker chapters.
- Each member of the group has Docker up and running on their machine.
- Each member of the group has a GitHub login that utilizes their school email.
- The group has designed and submitted a Project Proposal.
- The group has created a GitHub repository with all of the members as collaborators.

Two

- Each member of the group has read the Messaging, Database, Front End, and Back End chapters.
- The group has created a `docker-compose.yml` file in the group repository that brings up a messaging service, database service, front end service, and back end service.
- **Front End** has a "Hello World" page available on a port accessible from the local host.
- **Back End** can read from and write to a queue on **Messaging**.
- The completion of this milestone is documented through the use of Issues in the group GitHub repository.

Three

- **Back End** can read and write from **Database**.
- **Front End** has a register new user page and a login page. They do not have to be functional, but the

HTML for the page is being served.

- **Database** has a `user` table in a database on a persistent volume and the database can be accessed by **Back End**.
- Documentation is developed that describes how to use the RabbitMQ management interface to check to see if queues are being created. The documentation is in the form of a `README` file in any GitHub supported format within the messaging directory.
- The completion of this milestone is documented through the use of Issues in the group GitHub repository.

Four

- Each member of the group has read the Replication and Kubernetes chapters.
- Each member of the group has minikube up and running on their machine.
- **Database** has moved from a single instance to a replicated, high availability, load balancing cluster. For now, the instances can be statically configured inside a `docker-compose.yml` file.
- The completion of this milestone is documented through the use of Issues in the group GitHub repository.

Five

- Each member of the group has read the Database in Kubernetes and Messaging in Kubernetes chapters.
- **Database** has been migrated to Kubernetes using the minikube environment. All of the Kubernetes objects for **Database** are in one file named db-k8s.yml in the group GitHub repository.
- **Messaging** has been migrated to Kubernetes using the minikube environment. All of the Kubernetes objects are in one file named messaging-k8s.yml in the group GitHub repository.
- The completion of this milestone is documented through the use of Issues in the group GitHub repository.

Six

- Each member of the group has read the Front End in Kubernetes and Back End in Kubernetes chapters.
- **Front End** has been migrated to Kubernetes using the minikube environment. All of the Kubernetes objects are in one file in the named front-end-k8s.yml in the group GitHub repository.
- **Back End** has been migrated to Kubernetes using the minikube environment. All of the Kubernetes objects are in one file named back-end-k8s.yml in the group GitHub repository.
- The completion of this milestone is documented through the use of Issues in the group GitHub repository.

Deliverables

Deliverables are larger assessments designed to show a running system. Given the containerized nature of the project, it should be easy to bring up the deliverables on any system to test them. Groups are encouraged to test their system on different machines to make sure that everything will go well when it is time to assess their project. Groups are also encouraged and *expected* to keep the deliverables in mind over the course of the entire project.

Midterm

- **Front End** (Python, PHP, or Node) interacts with the user via HTTP and communicates with **Messaging** via a messaging library.
- **Messaging** (RabbitMQ) brokers the exchange of information between **Front End** and **Back End**.
- **Database** (PostgreSQL, MariaDB, or MySQL) is used by **Back End** for the storage of persistent information. All database files are stored in a Docker volume.
- **Back End** gathers information from data sources, stores information on **Database**, and interacts with **Messaging**.
- These four services are working with each other to provide a registration and authentication system for users.
- The project is fully testable on any machine running Docker by cloning the group git repository and running `docker-compose up` in the root of the project.

Final

- **Front End**, **Database**, **Messaging**, and **Back End** are all running in pods on a minikube Kubernetes Cluster. Each has three replicas.
- **Front End**, **Database**, **Messaging**, and **Back End** are all able to scale horizontally and recover from the failure of pods.
- The project is fully testable on any machine running minikube by cloning the group git repository, building the custom images locally (with an environment configured for the Docker daemon running *within* minikube), and running `kubectl apply -f .` in the root of the project.

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 cooperatively change a group of files and revert to a previous state if needed.

The benefits of a version control system can be more readily understood if we consider a few examples:

Example 2. 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 see what you changed and even roll-back your changes.

Example 3. Working with a Team

> You are working working with a team of people to build 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.[1] In the case of two people working on the same thing at the same time, a version control system could help with merging their changes by examining the lines that where changed, finding conflicts, and suggesting resolutions.

1.2. Installation

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

Windows

- git for windows: Installs git, git BASH, and a GUI. The git command is put in your path and can 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 configuration.

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 do that using any supported protocol.

1.3.3. Tracking / Staging Files

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

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. If you don't want to use an editor, the `-m` option allows you to specify a commit message on the command line.

Figure 1. Staging and Committing Changes

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 as 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 a common invocation 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 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 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 work.

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?*

   Version control allows you to keep track of *all* changes to a set of files. If something goes wrong and you need to get to a previous state, you can. Likewise, if you want to see how you did something in the past you can look through a history of changes. Finally, version control makes it easier for teams to work together by helping resolve conflicts when two authors change the same file.

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

   These are files that have been marked to be included in the next commit. Typically they have changes in them and they are part of a larger change to the entire project. They aren't committed yet, but once you perform the commit action and enter a commit message they will be.

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

`git add` can tell git to begin tracking a file *and* it can tell git to *stage* a file for the next commit.

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

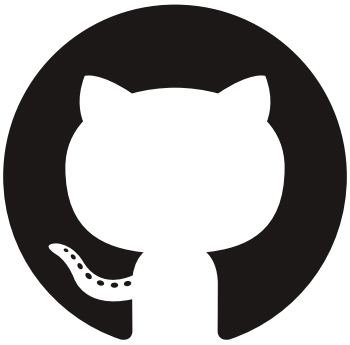
   The `git init` command turn the working directory into a git repository. Note that you need to be in the directory that you want to make into a repository.

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

   A remote is a non-local server (typically external) that stores your git repository. The `git push` command *pushes* the current version of your git repository to the remote server. It is usually used after you have completed a commit so that the other members in your team have access to the new version of the repository.

[1] It is important to note that although this is the dominant way git is used, it is not actually the way git was *intended* to be used. With git you can work in a group, merging changes from multiple contibuters, all without a central server.

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 service 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. It is recommended that you sign up with your .edu email address. This way you will have an *academic* account to showcase your work.

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 various tools may have trouble with them.
2. Sign in to GitHub and navigate to Create a New Repository
3. Put in the repository name (make sure it matches the 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. If you choose to use a private repository, be sure to add your instructor as a collaborator. Later when you want to showcase your work you can make this repository public. You could also start with a public repository which can provide good practice for learning to keep secrets (passwords, API keys, etc.) out of your commits. In this case you will still need to add group members as collaborators, but your instructor should have read-only access without any additional setup.
5. 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.
6. 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.

7. 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*. It can be found 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 project collaborator and it can be closed when it is resolved. Issues can also be linked to a milestone, which can be thought of as as group of things that need to be done to reach a particular phase.

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

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:

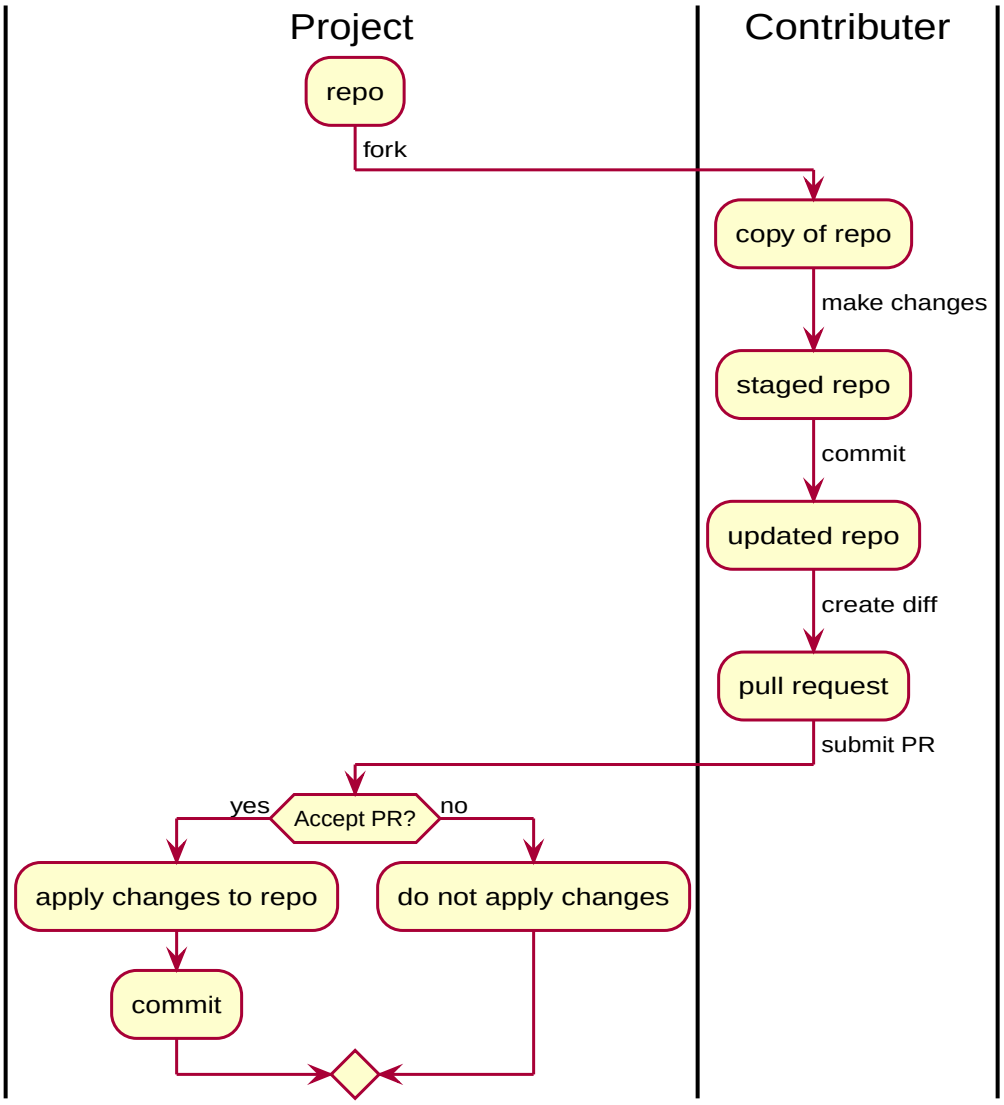


Figure 2. Pull Request

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.[2]

2.6. Resources

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

2.7. Questions

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

   GitHub provides a remote repository and several project management tools including an issue tracker. Both the repository and the tools have built-in web interfaces.

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

   git is the version control system and GitHub is a website that provides additional project management tools and a web interface for a git repository. git can be used independently of 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?*

   The team member will need to set up a GitHub account and tell you what their user name is. Once they have done that, you will need to add them as a collaborator to the repository that you want to share with them.

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

   Issues allow you to keep track of bugs that need to be fixed, milestones that need to be completed, or other general TODO items. It also gives you a place to have a dialog about the project you are working on.

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

   Even though all changes in a git repository can be rolled back if needed, it can be time consuming and annoying for a large project to have to keep reverting changes. By having contributors submit PRs, an integration team can review the changes and only commit code that meets their standards.

[2] If you're looking to take things a bit further AsciiDoc, reStructuredText, and scribble are worth exploring too. This book was written using 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 JavaScript Object Notation (JSON), but not as complex as Extensible Markup Language (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 file could have multiple *YAML 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"`.

YAML strings can use single, `'`, or double, `"`, quotes. Double quotes support escape sequences: `\n`, `\t`, `\"`, etc.

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 ISO8601 date elements.

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.

Learn to use | and |- for multiline strings. |- will do the same thing as | (as explained above), but will not add the newlines (everything will be on one line).

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 without the lines becoming too wide.
- 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 lifesaver when remotely logged in to a machine. 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. That being said, the learning curve is steep. If you are interested in learning vi, you may want to start with either vimtutor (packaged with vim) 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) 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
- Learn X in Y minutes Where X=yaml

3.5. Questions

1. *How does YAML signify different blocks?*

   Blocks are signified through indentation in YAML. This is similar to how Python delineates blocks.

2. *Are nested structures possible in YAML? Give an example.*

   Yes.

   ```
   level-1:
       level-1A:
       level-1B:
       level-1C:
   level-2:
       level-2A:
       level-2B:
       level-2C:
   ```

3. *What are the two components of a YAML mapping? Give an example to illustrate your point.*

   A key (or a keyword) and a value.

   ```
   key: value
   ```

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

   The # character at the beginning of a line comments out the whole line.

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

   This option inserts a series of spaces every time the tab key is pressed. \ This is important because YAML uses indentation to define blocks and the \ characters used to indent need to be consistent. With this option you can \ use the tab key and not have to worry about whether you are inserting a \ tab or a space.

Chapter 4. Docker

4.1. Purpose

Traditionally, virtualization has made use of a virtual machine (VM) to provide an isolated environment in which an operating system (OS) and applications can run. This allows the application to have a completely custom environment and ultimately makes it easier to deploy.

Example 4. Maintaining a Legacy Application

> Prithi is in charge of maintaining a payroll system that uses Perl, a web server, and a CGI module. This system requires specific, older versions of each of those components to function. Unfortunately all of the web servers that her company uses are upgrading to newer versions and removing Perl and the CGI module from their environment.
>
> In order to continue running the system, Prithi asks the team that maintains the servers if they will run a VM for her. Prithi can install the version of Linux that she needs on the VM and use the packages the payroll system requires. If appropriately isolated, any problems on the VM would be limited to just the payroll system. The web servers can be upgraded *and* Prithi can maintain the payroll system thanks to virtualization with VMs.

If you expand this example a bit, and have a team that runs a datacenter, provides VMs to customers, and charges for the amount of resources the VM uses, you have the beginnings of infrastructure as a service (IaaS). As servers run multiple VMs, you end up with a system that looks like this:

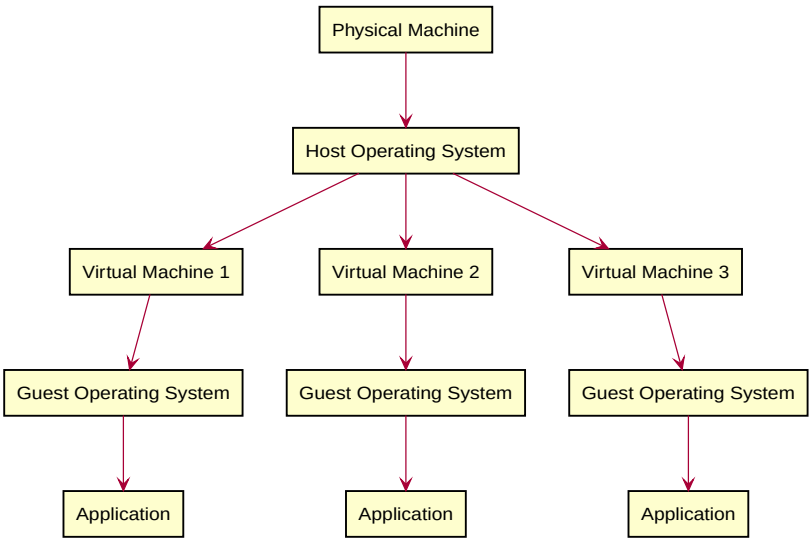


Figure 3. VM Server Architecture

Every VM having to run its own OS creates and emulate a unique machine creates a lot of overhead.

Containers allow for similar isolation as VMs, but at a reduced resource cost since. As opposed to VMs which emulate a physical machine, containers use control groups, namespaces, and chroot to allow *containers* to share the same operating system, but still be isolated. A server running containers has significantly less overhead and looks like this:

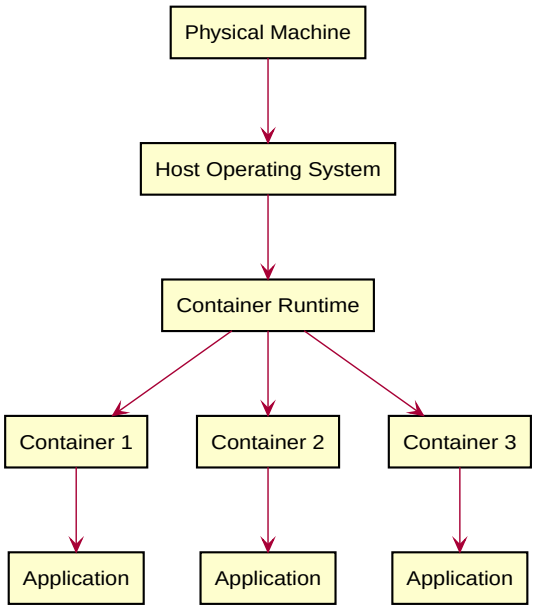


Figure 4. Container Server Architecture

There are many different container runtime environments including: LXC, containerd, rkt, and Docker (on containerd at this point). We will be using Docker due to its popularity and its ability be easily installed on various operating systems. By using Docker, we will be able to create a system that can be tested and used on any other machine that runs Docker.

4.2. Installation

There are several options for installing Docker on your device, depending on the operating system and hardware that you use:

Windows

- Docker Desktop should run fine on any Windows system that has Hyper-V (a virtualization feature) available.
- Windows 10 Home *does not* have Hyper-V available, but you can easily Upgrade to Windows 10 Education (typically this is provided by your University) and then instal it. This should be your first step if Docker Desktop does not work out-of-the-box.
- Docker Toolbox is an older version of Docker, running under a VM in VirtualBox. It can be difficult to work with and Docker Desktop is preferred.

Mac

- Docker Desktop should run fine on most Macs. We will be running it from the terminal so do not be alarmed if you install it and do not see an application running.

- If for some reason your computer is not supported you can try the MacOS version of Docker Toolbox, but Docker Desktop is preferred.

Linux

- Docker can easily be installed natively on Linux and packages for Docker exist for all major distributions.

4.3. Concepts

An image is the complete file system of a Linux instance. You can think of it like the hard drive of a server that someone has already set up. Images can be tagged using the format `name:version`. One of the reasons Docker is so popular is because there are many pre-built images available. If you try to use an image that does not exist on the local machine, Docker will automatically attempt to `pull` that image from the Docker Hub container registry.

Docker can `run` containers by taking images, copying the file system, and running commands on that file system within an isolated environment. This is called running a container. You can run several containers from the same image since the file system is cloned.

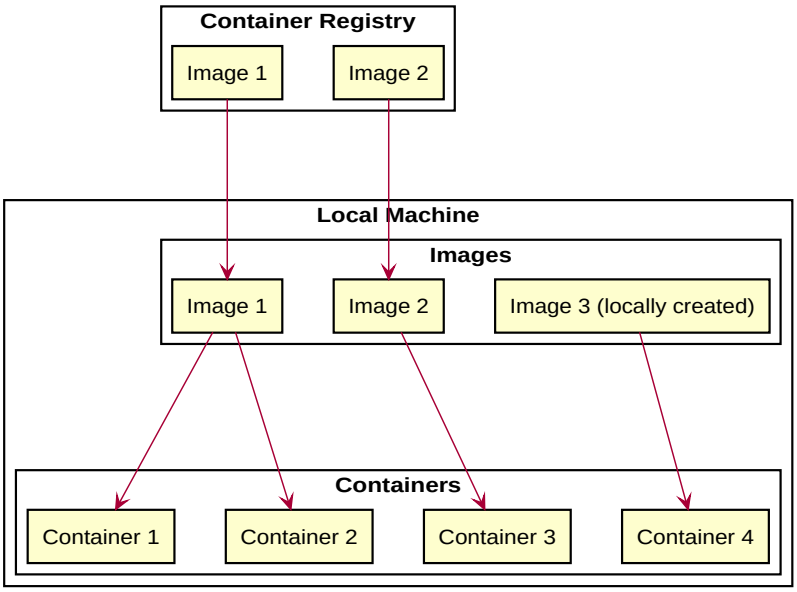


Figure 5. Images, Containers, and the Registry

In the above image, you can see that two images were pulled from the container registry: Image 1 and Image 2. Once images are pulled, they are locally cached. Image 3 was built locally and is available on the local machine, but is not available from the registry. From those images, four containers have been run. Container 1 and Container 2 are both using the same image, but keep in mind they have their own *copy* of the file system.

4.4. Commands

Docker commands are run via the `docker` command line interface (CLI) in a terminal. Here is a *brief* listing of some of the most useful commands:

images

Lists images that are locally available.

ps

Lists the *currently running* containers.

pull

Downloads an image from the container registry (Docker Hub by default).

run

Creates a container from an image and starts that container. If a command is specified that command is run, otherwise the default command for the image is run.

exec

Executes a command on an *already running* container.

stop

Stops a running container.

rm

Removes a container.

image rm

Removes an image.

build

Builds the Dockerfile in the directory specified into an image.

It is easy to forget that the build command takes an option as that option is sometimes the current directory: `.`. Don't forget the period at the end of a `docker build .` command.

Both the `run` and `exec` commands need to be passed the `-it` option if you want to run something interactively. This is often the case if you run a command like `bash` where you will by typing in shell commands.

As you use Docker, it will download / create a lot of resources and it can be helpful to clean those resources up periodically. Here are some commands to do just that:

system prune

Removes resources that aren't in use by any containers. This includes inactive containers: stopped containers that may be used again.

image prune --all

Removes all images that aren't *currently* being used. After you have been running Docker for a while, this can free up gigabytes (GB) of space.

4.5. Examples

For the following examples, it is assumed that you have Docker installed and that you are in a terminal with an environment that can run the `docker` command. This may be PowerShell if you installed Docker Desktop on Windows, Terminal if you are in MacOS, or even the Docker Quickstart terminal if you installed Docker Toolbox on Windows.

4.5.1. Running a Web Server

Let's take a look at just how easy it is to run a web server in a Linux container with Docker:

```
PS > docker run -d -p 8080:80 httpd:2.4.43 ①
Unable to find image 'httpd:2.4.43' locally
2.4.43: Pulling from library/httpd ②
54fec2fa59d0: Pull complete
8219e18ac429: Pull complete
3ae1b816f5e1: Pull complete
a5aa59ad8b5e: Pull complete
4f6febfae8db: Pull complete
Digest: sha256:c9e4386ebcdf0583204e7a54d7a827577b5ff98b932c498e9ee603f7050db1c1
Status: Downloaded newer image for httpd:2.4.43

fa13023485993c9ec47c805d0ce06b69b305ddf61657fbb6ec58674abb5a057b ③
```

① Here we are telling Docker we want to `run` a container , `-d` in the background (daemon mode), `-p 8080:80` with port 8080 on our local machine forwarded to port 80 on the container, and the image is version `2.4.43` of `httpd`. You can read more about this image here.

② Because we don't have the image locally, it is pulled from the Docker Hub container registry.

③ Each running container is given a unique hash.

Now let's see what containers we have running with the `docker ps` command:

```
PS > docker ps ①
CONTAINER ID        IMAGE               COMMAND             CREATED             STATUS
PORTS                  NAMES
fa1302348599        httpd:2.4.43        "httpd-foreground"   5 seconds ago       Up 5
seconds        0.0.0.0:8080->80/tcp   eager_heyrovsky ②
```

① `docker ps` shows all *running* containers, to see *all* containers, including stopped containers, we could have used `docker ps -a`.

② This displays the container id (enough of the hash to identify it for commands), the image that the container is copied from, when it was started, what its status is, which ports are being forwarded, and a friendly name for the container (automatically generated if you don't specify one) that you can use in place of the container id in commands.

If you've been following along with these commands, you should be able to open a web browser, go to http://localhost:8080, and see a page stating, "It works!"

Now let's clean up:

```
PS > docker stop fa1302348599 ①
fa1302348599

PS > docker ps ②
CONTAINER ID        IMAGE               COMMAND             CREATED             STATUS
PS > docker ps -a ③
CONTAINER ID        IMAGE               COMMAND             CREATED             STATUS
fa1302348599        httpd:2.4.43        "httpd-foreground"  18 minutes ago      Exited
(0) 5 seconds ago

PS > docker rm fa1302348599 ④
fa1302348599

PS > docker ps -a
CONTAINER ID        IMAGE               COMMAND             CREATED             STATUS
```

① We know the id from the previous `docker ps` command, by running `docker stop` we can stop the container.

② Now we won't see it in the plain `docker ps` command.

③ It is still there, but stopped. We can see it with the `docker ps -a` command.

④ `docker rm` will remove the stopped container.

In most situations, `docker rm` isn't really necessary as stopped containers do not actively consume resources other than local storage. Try browsing to http://localhost:8080 again to confirm that the container is not running. You should be unable to connect.

4.5.2. Building a Custom Image

Sometimes the stock images are not enough to do what you want. In this example we will be building an image for a web server that still uses `httpd:2.4.43` as its base, but adds an extra file to change the default page. To do this, we need to create a directory with a `Dockerfile` and our new `index.html` page. We'll do this in the `docker-demo` directory:

docker-demo/index.html

```
<h1>Hello from a container running a custom image!</h1>
```

docker-demo/Dockerfile

```
FROM httpd:2.4.43
COPY index.html /usr/local/apache2/htdocs/
```

A Dockerfile is a list of basic instructions for building an image. Instructions are typically capitalized and in this case our Dockerfile is using two of them: `FROM` and `COPY`. `FROM` tells Docker that you want to build an image on top of another image. In this case we want to build our image on top of the `httpd:2.4.43` image. We also use the `COPY` instruction to copy a local file into the image we are building. In this case we copy our `index.html` into the directory that the Apache web server serves files from.

Details about what directories an image uses can often be found in the image's documentation on Docker Hub.

To build our Dockerfile into an image, we are going to use the `build` command:

```
PS docker-demo> docker build -t docker-demo:v1 . ①
Sending build context to Docker daemon  3.072kB
Step 1/2 : FROM httpd:2.4.43 ②
 ---> b2c2ab6dcf2e
Step 2/2 : COPY index.html /usr/local/apache2/htdocs/
 ---> Using cache
 ---> 7a8122895898
Successfully built 7a8122895898
Successfully tagged docker-demo:v1
SECURITY WARNING: You are building a Docker image from Windows against a non-Windows
Docker host. All files and directories adde
d to build context will have '-rwxr-xr-x' permissions. It is recommended to double check
and reset permissions for sensitive fil
es and directories. ③

PS docker-demo> docker images ④
REPOSITORY          TAG                 IMAGE ID            CREATED             SIZE
docker-demo         v1                  7a8122895898        17 minutes ago      166MB
httpd               2.4.43              b2c2ab6dcf2e        2 weeks ago         166MB
```

① Here we specify that we want to `build` an image and we want to tag it as `docker-demo:v1`. This command takes one argument, the directory which contains the Dockerfile. In our case we pass it `.` to signify the current directory.

② You can see each instruction and the results as they are performed.

③ Windows does not support the same file permissions as Linux so you may see this warning if you are building on Windows.

④ Finally we look at the images currently available. We should see that the one we built is available.

Now let's run our custom image in a container with the `docker run` command:

```
PS docker-demo> docker run -p 8080:80 docker-demo:v1 ①
AH00558: httpd: Could not reliably determine the server's fully qualified domain name,
using 172.17.0.2. Set the 'ServerName' directive globally to suppress this message
AH00558: httpd: Could not reliably determine the server's fully qualified domain name,
using 172.17.0.2. Set the 'ServerName' directive globally to suppress this message
[Sat May 09 15:09:48.016060 2020] [mpm_event:notice] [pid 1:tid 140614148256896] AH00489:
Apache/2.4.43 (Unix) configured -- resuming normal operations
[Sat May 09 15:09:48.016154 2020] [core:notice] [pid 1:tid 140614148256896] AH00094:
Command line: 'httpd -D FOREGROUND'
```

① Notice that we didn't pass the `-d` option to Docker run, meaning we are running in the foreground. This is useful if we want to run something quickly as the log messages are printed directly in your terminal.

If you open a web browser and navigate to https://localhost:8080 you should see the message "Hello from a container running a custom image!" When you are done, you can type Ctrl+C in the terminal to stop running the container, use `docker ps` to get its ID, and then use `docker stop` to stop it.

4.5.3. Using Docker Compose

Having to specify command line arguments for the `docker` command can get tedious, especially as your environment becomes more complex. Docker Compose is a tool bundled with Docker that can be used to store the configuration for a multi-container setup in a single `docker-compose.yml` file. The best way to learn about it is to see it in action, so let's start by looking at an example:

docker-demo/docker-compose.yml

```
version: '3'

services:
  web:
    build: .
    ports:
      - "8080:80"
```

This file defines a Docker Compose service named `web` that builds an image from the directory that `docker-compose.yml` is in. A container is run with the image and local port 8080 is forwarded to port 80 in the container.

Here is a brief listing of some of the most useful docker-compose commands:

up

Brings up all services in `docker-compose.yml`

down

Brings down all services in `docker-compose.yml`

stop

Stops all services or a service specified

exec

Runs a command on *running* service

logs

Prints the logs for a service

Now that we know a few commands, let's bring up the `web` service:

```
PS docker-demo> docker-compose up ①
Creating network "docker-demo_default" with the default driver ②
Building web
Step 1/2 : FROM httpd:2.4.43
 ---> b2c2ab6dcf2e
Step 2/2 : COPY index.html /usr/local/apache2/htdocs/
 ---> 6ac4e496ced0
Successfully built 6ac4e496ced0
Successfully tagged docker-demo_web:latest
WARNING: Image for service web was built because it did not already exist. To rebuild
this image you must use `docker-compose build` or `docker-compose up --build`. ③
Creating docker-demo_web_1 ... done
Attaching to docker-demo_web_1
web_1  | AH00558: httpd: Could not reliably determine the server's fully qualified domain
name, using 172.18.0.2. Set the 'Serve
rName' directive globally to suppress this message
web_1  | AH00558: httpd: Could not reliably determine the server's fully qualified domain
name, using 172.18.0.2. Set the 'Serve
rName' directive globally to suppress this message
web_1  | [Sat May 09 17:05:42.377243 2020] [mpm_event:notice] [pid 1:tid 140538714080384]
AH00489: Apache/2.4.43 (Unix) configur
ed -- resuming normal operations
web_1  | [Sat May 09 17:05:42.377338 2020] [core:notice] [pid 1:tid 140538714080384]
AH00094: Command line: 'httpd -D FOREGROUND
'
web_1  | 172.18.0.1 - - [09/May/2020:17:05:57 +0000] "GET / HTTP/1.1" 304 - ④
```

① Remember this command brings up *all* the services in the `docker-compose.yml` file in the *current directory*.

② Docker Compose also creates a network for your services to run in. This provides isolation for your services and helps with service discovery.

③ This often trips up people the first time they use Docker Compose. If you make changes to a Dockerfile you will have to rebuild the image, or specify --build to `docker-compose up`, otherwise you won't see your changes.

④ Docker Compose runs in the foreground by default and displays the aggregate log information from all of the services running. Log messages are prefixed by the service name. This can be *very* useful for debugging.

We can test that things are working by going to http://localhost:8080. When we are all done we can close things down with by typing Ctrl+C in the terminal.

4.6. Resources

- Docker overview

- Docker Quickstart
- Best practices for writing Dockerfiles
- Dockerfile reference
- Overview of Docker Compose

4.7. Questions

1. *What is the difference between a VM and a container?*

2. *What is the difference between a Docker image and a Docker container?*

3. *How would you run a shell on an already running container?*

4. *What does the `-d` option do when passed to the `docker run` command? When may you want to use it? When may you not want to use it?*

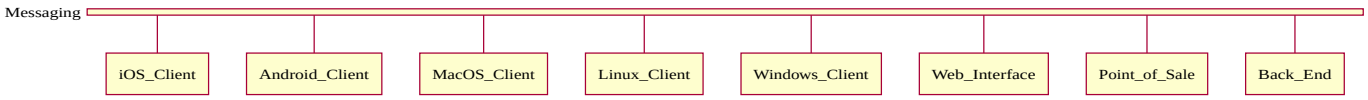
5. *What does Docker Compose do and how it is different from the `docker` command?*

Chapter 5. Messaging

5.1. Purpose

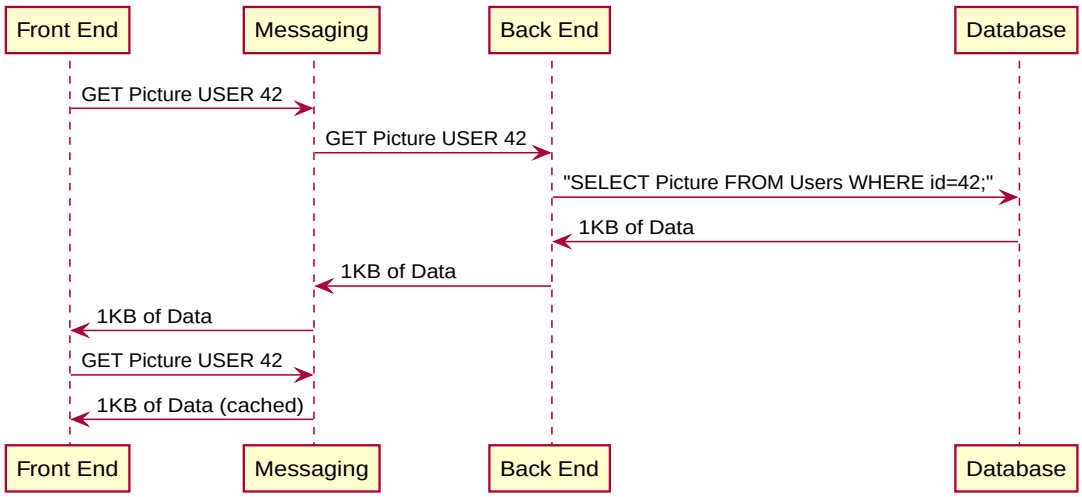
Messaging allows multiple components to use a common medium to exchange information. The goal is to create a system that is more extensible, resilient, and scalable. By using a messaging framework you can establish the interfaces between components and worry less about idiosyncrasies of communication.

One of the easiest ways to understand the advantages of a messaging layer is to view it as a bus carrying information between components. While you may only have a few components now, what happens if you want to expand?



It would be arduous to have to write unique code for eight components to facilitate communication. By using a messaging framework, this is taken care of.

If the interfaces are well thought out, messaging frameworks can also take advantage of caching. Examine the following scenario:



The second time a `GET Picture USER 42` request is received, **Messaging** already knows the answer and can respond with a cached copy. A well-designed messaging framework will allow you to adjust cache parameters to achieve significant performance gains without serving stale data. Cache invalidation is a

famously difficult problem, but at least with a messaging framework you are not trying to solve it by yourself.

Messaging frameworks allow components to be built and profiled independent of other components. Want to know how many requests a client is generating so you can better build a back end to support it? Check the statistics of the messaging queues you are using. Want to be able to completely swap out a component without any of the other components noticing? Use a messaging framework.

That's not to say that all projects take advantage of a messaging layer. Often it's viewed as "just another thing to get in the way" or another possible bottleneck. Many projects attempt to shoe horn the messaging framework in later on, a difficult feat. We will avoid that here by mandating that *a messaging framework must be used from the start.*

5.2. Frameworks

The two most popular messaging frameworks are RabbitMQ and Kafka. We will briefly discuss each platform, but it is important to note that we will be using RabbitMQ in this text. Milestones and deliverables in the project will require things specific to RabbitMQ. All systems must function within constraints and one of the constraints on the system we are going to design is that it *must* use RabbitMQ.

RabbitMQ is built with Erlang and OTP, a system meant for solving the large-scale, distributed problems that are prevalent in the telecom industry. It is hard to think of a platform more suited to the task of messaging. RabbitMQ supports traditional messaging paradigms and uses a standard messaging protocol: AMQP. This allows for application libraries in *many* different languages including Python, Java, Ruby, PHP, C#, JavaScript, Go, Elixir, Objective-C, Swift, and Spring AMQP.

Kafka was built by the Apache project in Java. It uses the publish and subscribe model, the most common messaging model. It includes a broker by default meaning it often requires less set up for standard messaging scenarios. While it may not offer as much choice in *exactly* how messages are handled it does support large data streams incredibly well.

5.3. RabbitMQ and Docker

The documentation for the official RabbitMQ Docker image is quite good and covers many of the topics we will touch on in here in greater detail. Please refer to it for further explanation.

The stock RabbitMQ image uses different versions to specify if the management interface should be enabled. You will probably want to use the web-based management interface at first to see what is going on. To do so you will need to run a management version and you will also need to expose port `15672` on your container to your local machine.

RabbitMQ can also use a few different environment variables to store secrets:

RABBITMQ_DEFAULT_USER

By default, RabbitMQ will use the user name `guest`. Technically we should only be able access our RabbitMQ container internally, but it is still a good practice to change this to something specific to our project.

RABBITMQ_DEFAULT_PASS

By default, RabbitMQ will use the password `guest`. It would be a good idea to change this as well.

RABBITMQ_ERLANG_COOKIE

Erlang nodes use this for authentication. As far as we are concerned this will only become important when we run multiple instances of a RabbitMQ container and they need to communicate with each other.

Let's start up a container running the management interface and take a look:

```
PS > docker run --env RABBITMQ_DEFAULT_USER=example --env RABBITMQ_DEFAULT_PASS=example
-p 15672:15672 rabbitmq:3.8.3-management①
Unable to find image 'rabbitmq:3.8.3-management' locally ②
3.8.3-management: Pulling from library/rabbitmq
23884877105a: Pull complete
bc38caa0f5b9: Pull complete
<snip>
2020-05-02 20:45:37.812 [info] <0.278.0>
 Starting RabbitMQ 3.8.3 on Erlang 22.3.3 ③
 Copyright (c) 2007-2020 Pivotal Software, Inc.
 Licensed under the MPL 1.1. Website: https://rabbitmq.com

  ##  ##      RabbitMQ 3.8.3
  ##  ##
  ##########  Copyright (c) 2007-2020 Pivotal Software, Inc.
  ######  ##
  ##########  Licensed under the MPL 1.1. Website: https://rabbitmq.com

  Doc guides: https://rabbitmq.com/documentation.html
  Support:    https://rabbitmq.com/contact.html
  Tutorials:  https://rabbitmq.com/getstarted.html
  Monitoring: https://rabbitmq.com/monitoring.html

  Logs: <stdout>

  Config file(s): /etc/rabbitmq/rabbitmq.conf

  Starting broker...2020-05-02 20:45:37.814 [info] <0.278.0>
 node           : rabbit@53c4ff237fcd
 home dir       : /var/lib/rabbitmq
 config file(s) : /etc/rabbitmq/rabbitmq.conf
 cookie hash    : mCbpvRz5+sUotXe8uIyiKQ== ④
 log(s)         : <stdout>
 database dir   : /var/lib/rabbitmq/mnesia/rabbit@53c4ff237fcd
2020-05-02 20:45:37.827 [info] <0.278.0> Running boot step pre_boot defined by app rabbit
<snip> ⑤
2020-05-02 20:45:38.683 [info] <0.9.0> Server startup complete; 3 plugins started.
 * rabbitmq_management
 * rabbitmq_management_agent
 * rabbitmq_web_dispatch
 completed with 3 plugins. ⑥
```

① Notice how we specify a new user and password, forward a port from our local machine, and request the management image.

② Since we don't have the image, it will be pulled from Docker Hub automatically.

③ RabbitMQ tells you what version of Erlang it is using. Clustering requires nodes to run similar

versions, so this can be an important bit of information.

④ If you don't specify an Erlang cookie, one will be chose randomly. This is it, base64 encoded.

⑤ There are *a lot* of startup messages and it can take some time for RabbitMQ to start. I've cut them out of the output here.

⑥ Once you see this, you're node is up and running.

Now, we should be able to visit http://localhost:15672 and see the management interface running. Using our user name / password of example / example, we can sign in. Take a moment to look around the interface. Queues are created automatically by applications so you won't need to configure anything, but once you have components up and using RabbitMQ you can look here to see the queues they've created and to make sure that everything is working. When you are all done type Ctrl+C to detach from the `docker run command` and use `docker stop` to stop the container:

```
PS > docker ps ①
CONTAINER ID        IMAGE                          COMMAND                  CREATED
53c4ff237fcd        rabbitmq:3.8.3-management   "docker-entrypoint.s…"   12 minutes ago
PS > docker stop 53c4ff237fcd ②
53c4ff237fcd
PS > docker ps ③
CONTAINER ID        IMAGE               COMMAND             CREATED             STATUS
```

① List all running containers

② Stop rabbitmq by ID

③ Double-check to make sure nothing is running

5.4. Resources

- AMQP
- RabbitMQ Getting Started
- RabbitMQ Docker Hub Image
- Docker Networking Overview

5.5. Questions

1. *Why would a project choose to use a messaging framework?*

2. *What is caching and what are its benefits?*

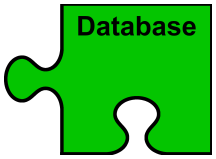
3. *What is AMQP?*

4. *How can you specify that you want to enable the management interface in the official RabbitMQ Docker image?*

5. *Why should you change the RabbitMQ default password and how do you change it?*

6. *How would you start up a RabbitMQ instance in Docker Compose, with the same options we used in the example?*

Chapter 6. Database



6.1. Introduction

Databases are used to store and organize information in a manner consistent with the relationships in that data. This allows for more natural queries to retrieve information. For example, if you typically look up a car part by its ID, it would make sense to have a table of part IDs with names and descriptions. A car can be thought of as a collection of parts, therefore you could have a car table and a car_part table that relates a car ID to multiple part IDs:

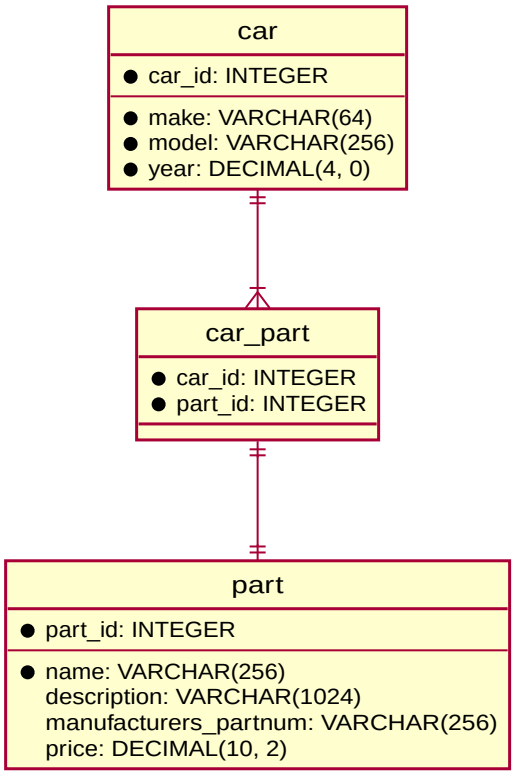


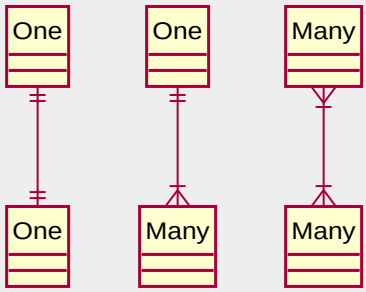
Figure 6. Tables in a Relational Database

Entity Relationship (ER) Diagrams

ER diagrams in this text are based on the Information Entineering notation as supported by PlantUML:

- leading dots are used to represent mandatory (NOT NULL) attributes
- primary attributes are listed immediately under the entity name
- other attributes are listed following the primary attributes
- types (in SQL syntax) follow the `:` on the attribute line

Lastly, crows feet are used to represent the arity of relations:



Relational databases maintain the integrity of their relations at the cost of being rigid and requiring more setup. In our system we will employ a relational database for multiple reasons:

- Relational databases are the most popular databases in use today.
- SQL is the most marketable computer language in the world and with the growth of data science, this is not changing any time soon.
- Relational databases present unique challenges with regard to replication. This will provide a good learning opportunity later in the text.

A database will be used for the *persistent* storage of information. In a container environment, it is very important to make the distinction between *ephemeral* and *persistent* storage. A container may have lots of information on its filesystem but if that container is stopped, that information is lost. All persistent information, that is information that needs to survive the container lifecycle, should be stored on a database which utilizes an external volume. Having persistent storage needs makes a container *stateful*.

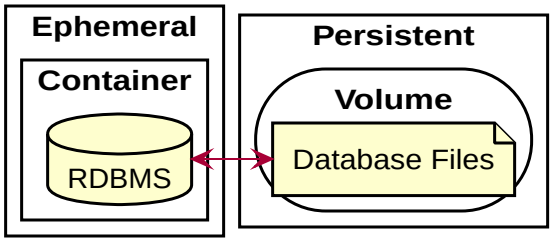


Figure 7. Containerized Database using a Volume

As can be seen above, the volume exists *outside* of the container running the relational database management system (RDBMS). The volume holds the database files the RDBMS uses to store information and if the container is stopped or removed, the files remain.

6.2. Popular RDMS

For our project we can use any one of the following RDMS:

MySQL

MySQL is an RDMS that has grown out of the practical need for a relational database in Linux. It quite literally puts the 'M' in the venerable LAMP stack (Linux Apache MySQL PHP). It has native support for replication and many other practical features that make it an excellent choice for any project.

MariaDB

MariaDB is a RDMS that aims to be be feature and syntax compliant with MySQL. It was originally forked from MySQL and developed as a response to Oracle's purchase of Sun Microsystems, who owned MySQL. It as actively developed and widely popular.

PostgreSQL

PostgreSQL is an *object* relational database management system that aims to be more SQL compliant than MySQL. Where MySQL was borne out of workplace need, PostgreSQL has an academic lineage. It is enterprise ready, massively scalable (although it relies on several other tools for replication), and has fewer licensing issues than MySQL. PostgreSQL is seeing growing adoption.

6.3. Example

In this example we will start a MySQL container, initialize it with the SQL file shown below, and connect to it with the mysql client. The `setup.sql` file that initializes our database can be found in the `db-demo` directory. It sets up a simple car part database for our example:

db-demo/setup.sql

```
CREATE TABLE car (
    car_id INTEGER,
    make VARCHAR(64) NOT NULL,
```

```
    model VARCHAR(256) NOT NULL,
    year DECIMAL(4, 0) NOT NULL,
    PRIMARY KEY(car_id)
);

CREATE TABLE part (
    part_id INTEGER,
    name VARCHAR(256) NOT NULL,
    description VARCHAR(1024),
    manufacturers_partnum VARCHAR(256),
    price DECIMAL(10, 2),
    PRIMARY KEY(part_id)
);

CREATE TABLE car_part (
    car_id INTEGER,
    part_id INTEGER,
    FOREIGN KEY(car_id) REFERENCES car(car_id),
    FOREIGN KEY(part_id) REFERENCES part(part_id),
    PRIMARY KEY(car_id, part_id)
);

INSERT INTO car VALUES (0, 'HONDA', 'CIVIC', 2005);
INSERT INTO car VALUES (1, 'TOYOTA', 'COROLLA', 2010);
INSERT INTO car VALUES (2, 'FORD', 'F-150', 2009);
INSERT INTO part
VALUES (0, 'Brake Pads', 'Duralast ceramic brake pads', 'MKD621', 19.99);
INSERT INTO part
VALUES (1, 'Alternator', 'Duralast gold new alternator', 'DLG12308', 175.99);
INSERT INTO part
VALUES (2, 'Radiator Cap', 'Duralast Radiator Cap', '7036', 10.99);
INSERT INTO part
VALUES (3, 'Alternator', 'Duralast gold new alternator', 'DLG5540-17-2',
    256.99);
INSERT INTO part
VALUES (4, 'Rear Leaf Spring', '1500lbs capacity', '43-1781', 129.99);

INSERT INTO car_part VALUES (0, 0);
INSERT INTO car_part VALUES (0, 1);
INSERT INTO car_part VALUES (0, 2);
INSERT INTO car_part VALUES (1, 2);
INSERT INTO car_part VALUES (1, 3);
INSERT INTO car_part VALUES (2, 2);
INSERT INTO car_part VALUES (2, 4);
```

Be sure to check the documentation for the Docker image you are using to see how to initialize a database and to learn about which environment variables you will need to use. In this case, we will

bind mount the current directory to `/docker-entrypoint-initdb.d/` on the container so our `setup.sql` file is run the first time the container is started. We will also use environment variables to set up a database, username, and password.

Now let's create a volume, and run our container:

```
PS db-demo> docker volume create db-data ①
db-data
PS db-demo> docker volume ls ②
DRIVER              VOLUME NAME
local               db-data
PS db-demo> docker run -d --mount "type=volume,src=db-data,dst=/var/lib/mysql" --mount
"type=bind,src=$(pwd),dst=/docker-entrypoint-initdb.d" -e MYSQL_ROOT_PASSWORD=changeme -e
MYSQL_DATABASE=cars -e MYSQL_USER=car_user -e MYSQL_PASSWORD=changeme mysql:8.0.20 ③
32193c2a74a16f73c44dc186a9b368fafd4b0d1fadc000c58ee8e3357fc24a6e
PS db-demo> docker ps ④
CONTAINER ID        IMAGE               COMMAND                  CREATED
STATUS              PORTS                 NAMES
32193c2a74a1        mysql:8.0.20        "docker-entrypoint.s…"   12 seconds ago      Up
12 seconds       3306/tcp, 33060/tcp   youthful_euler
PS db-demo> docker logs 32193c2a74a1 ⑤
2020-05-11 17:04:07+00:00 [Note] [Entrypoint]: Entrypoint script for MySQL Server 8.0.20-
1debian10 started.
2020-05-11 17:04:07+00:00 [Note] [Entrypoint]: Switching to dedicated user 'mysql'
2020-05-11 17:04:07+00:00 [Note] [Entrypoint]: Entrypoint script for MySQL Server 8.0.20-
1debian10 started.
2020-05-11 17:04:07+00:00 [Note] [Entrypoint]: Initializing database files
<snip>
2020-05-11 17:04:17+00:00 [Note] [Entrypoint]: Creating database cars
2020-05-11 17:04:17+00:00 [Note] [Entrypoint]: Creating user car_user
2020-05-11 17:04:17+00:00 [Note] [Entrypoint]: Giving user car_user access to schema cars

2020-05-11 17:04:17+00:00 [Note] [Entrypoint]: /usr/local/bin/docker-entrypoint.sh:
running /docker-entrypoint-initdb.d/setup.sql ⑥


2020-05-11 17:04:18+00:00 [Note] [Entrypoint]: Stopping temporary server
<snip>
2020-05-11T17:04:21.787429Z 0 [System] [MY-010931] [Server] /usr/sbin/mysqld: ready for
connections. Version: '8.0.20'  socket: '/var/run/mysqld/mysqld.sock'  port: 3306  MySQL
Community Server - GPL.
```

① We create a volume named `db-data`

② and now we can see it listed in the available volumes.

③ This lengthy command translates to:

- `run`: mysql version 8.0.20
- `-d`: in the background
- `--mount "type=volume,src=db-data,dst=/var/lib/mysql"`: mount the volume named db-data to /var/lib/mysql on the container
- `--mount "type=bind,src=$(pwd),dst=/docker-entrypoint-initdb.d"`: mount the current directory, we have to use pwd here because it wants an absolute path, to /docker-entrypoint-initdb.d on the container. This is a bind mount which is a quick linkage between a directory on the host and a directory on the container. Bind mounts may not support all of the features you need, but for our purposes it works well here. Our current directory has the `setup.sql` file that we use to initialize our database.
- `-e MYSQL_ROOT_PASSWORD=changeme`: use an environment variable to set the root password for mysql access
- `-e MYSQL_DATABASE=cars`: use an environment variable to set the database we initialize
- `-e MYSQL_USER=car_user`: use an environment variable to create a new user that can access the database we set up
- `-e MYSQL_PASSWORD=changeme`: use an environment variable to set a password for the user we created above

④ Check to see that our container is running and get the ID

⑤ Examine the log output for our container (by ID)

⑥ We can see here in the output, that our `setup.sql` script was run

With our container running in the background, we can now connect and make sure that our database was initialized. To avoid having to download a mysql client on the host, we will `exec` the mysql command on the running container:

```
PS db-demo> docker exec -it 32193c2a74a1 mysql -u car_user -p cars
Enter password: ①
Reading table information for completion of table and column names
You can turn off this feature to get a quicker startup with -A

Welcome to the MySQL monitor.  Commands end with ; or \g.
Your MySQL connection id is 12
Server version: 8.0.20 MySQL Community Server - GPL

Copyright (c) 2000, 2020, Oracle and/or its affiliates. All rights reserved.

Oracle is a registered trademark of Oracle Corporation and/or its
affiliates. Other names may be trademarks of their respective
owners.

Type 'help;' or '\h' for help. Type '\c' to clear the current input statement.
```

```
mysql> show tables; ②
+---------------+
| Tables_in_cars |
+---------------+
| car            |
| car_part       |
| part           |
+---------------+
3 rows in set (0.00 sec)

mysql> select * from car; ③
+--------+--------+---------+------+
| car_id | make   | model   | year |
+--------+--------+---------+------+
|      0 | HONDA  | CIVIC   | 2005 |
|      1 | TOYOTA | COROLLA | 2010 |
|      2 | FORD   | F-150   | 2009 |
+--------+--------+---------+------+
3 rows in set (0.00 sec)

mysql> SELECT name, description, price
    -> FROM car, car_part, part
    -> WHERE car.car_id=car_part.car_id AND
    -> part.part_id=car_part.part_id AND
    -> car.make='FORD' AND car.model='F-150' AND car.year='2009'; ④
+-----------------+-----------------------+--------+
| name            | description           | price  |
+-----------------+-----------------------+--------+
| Radiator Cap    | Duralast Radiator Cap |  10.99 |
| Rear Leaf Spring | 1500lbs capacity      | 129.99 |
+-----------------+-----------------------+--------+
2 rows in set (0.00 sec)
```

① recall the we set the password to `changeme` via an environment variable

② check that all the tables were created

③ check that the data was inserted into the tables

④ perform a sample query looking up all parts for a 2009 Ford F-150

`docker volume prune` is a useful command for getting rid of volumes not in use by any container. Just be sure that you don't need the data on those volumes anymore!

6.4. Resources

- MySQL Docker Image

- MariaDB Docker Image
- PostgreSQL Docker Image
- Creating Users / Granting Privileges in MySQL/MariaDB
- Volumes in Docker
- MySQL Connector Python Module
- Psycopg Python Module

6.5. Questions

1. *What is the difference between persistent and ephemeral data?*

2. *What is a Docker volume and why should a containerized RDMS use it?*

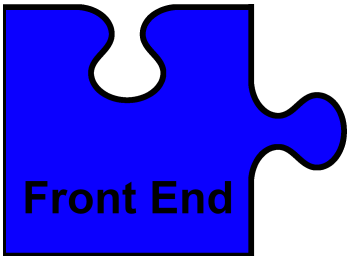
3. *Pick one RDMS from the list above. What are the advantages of this particular RDMS over the others?*

4. *How do you set an environment variable via the* `docker` *command?*

5. *How do you initialize a database in the default MySQL docker image?*

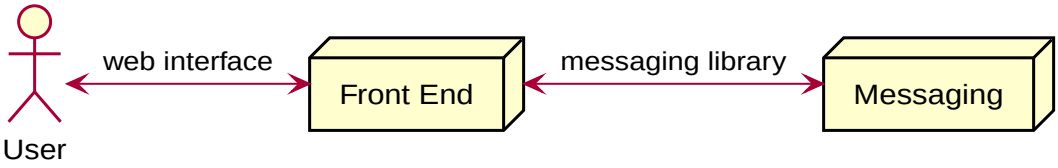
6. *The command we used to run our database container was very complex and it would be quite tedious to type it out over and over. How would you run the example in Docker Compose instead?*

Chapter 7. Front End



7.1. Introduction

The purpose of a front end is to interact with the user. Web applications are applications that use a web server to present their front end. In our case, we will build a web application that interacts with the user via a web server, but uses **Messaging** for any other communication:



To provide the web interface, most applications use some sort of web framework. The decision of which web framework to use depends largely on the language(s) the **Front End** developer is comfortable with and what features they would like. The most common web frameworks for various languages are shown below. You are encouraged to research them and determine what would best fit your needs.

Python

- Flask
- Django
- Bottle

JavaScript

- Plain NodeJS
- Express
- Koa

PHP

Largely employed as a templating language, PHP is still popular and can be used in conjunction with a web server as a framework.

To communicate with **Messaging** a library is used. Ultimately this will allow for a more scalable application. The decision of which library to use is mostly dependant on what language **Front End** is

using. Fortunately RabbitMQ has extensive documentation on using the most popular messaging libraries for various languages:

Python	pika
JavaScript	amqp
PHP	php-amqplib

A **Front End** developer should be comfortable with HTML for the creation of views that the user will see and they should be mindful that at some point their code will need to be run on a production server. There are many options for production servers depending on what language **Front End** is using:

Python	Frameworks use WSGI for which there are many options.
JavaScript	Frameworks make use of Node and typically use managed node processes.
PHP	Usually served using CGI or a module for a standard web server.

7.2. Example

As an example, we are going to set up a PHP application that presents a web interface *and* communicates with **Messaging**. We will be basing our code off of the RabbitMQ RPC tutorial for PHP and using the official PHP Docker Hub image.

The first problem we have to solve is how to include php-amqplib with our application. To do this, we will follow php-amqplib's recommendation and use composer. No matter what language you are using for **Front End** it is *very* important that you learn to use its package management system. To start, we create a `composer.json` file that lists what version of PHP we wish to build for, what PHP extensions are needed, and what our application's requirements are:

front-end-demo/web/composer.json

```
{
    "config": {
        "platform": {
            "php": "7.4.5",
            "ext-bcmath": "1",
            "ext-sockets": "1"
        }
    },
    "require": {
        "php-amqplib/php-amqplib": ">=2.9.0"
    }
}
```

The PHP extensions `bcmath` and `sockets` are both required by `php-emqplib`. We request them in the composer file, but will also install them in the `Dockerfile`

Now we need to use composer to download and install packages in the `vendor` directory of our application. Fortunately the official composer Docker image makes this easy. We can run composer on the image and store the output through careful use of a bind mount:

```
PS front-end-demo\app> docker run --rm -it -v "$(pwd):/app" composer install ①
Loading composer repositories with package information
Updating dependencies (including require-dev)
Package operations: 2 installs, 0 updates, 0 removals
  - Installing phpseclib/phpseclib (2.0.27): Downloading (100%)
  - Installing php-amqplib/php-amqplib (v2.11.2): Downloading (100%)
phpseclib/phpseclib suggests installing ext-libsodium (SSH2/SFTP can make use of some
algorithms provided by the libsodium-php extension.)
phpseclib/phpseclib suggests installing ext-mcrypt (Install the Mcrypt extension in order
to speed up a few other cryptographic operations.)
phpseclib/phpseclib suggests installing ext-gmp (Install the GMP (GNU Multiple Precision)
extension in order to speed up arbitrary precision integer arithmetic operations.)
Writing lock file
Generating autoload files
1 package you are using is looking for funding.
Use the `composer fund` command to find out more!
```

① The `--rm` option removes the container after it finishes. Notice how we bind link the app directory `front-end-demo/app` (which we are in) with the `/app` directory on the container. This way all the files that composer creates are stored locally in our app directory.

We will also need a PHP image with some extra PHP extensions enabled and our application code in the `/var/www/html` directory. To do this, we use a simple Dockerfile:

front-end-demo/Dockerfile

```
FROM php:7.4.5-apache
RUN docker-php-ext-install bcmath sockets
COPY app/ /var/www/html/
```

Since we are using a container for **Messaging** as well as one for **Front End**, it would be easiest to use Docker Compose to bring them both up at the same time. For that, we need to create a `docker-compose.yml` file:

front-end-demo/docker-compose.yml

```
version: "3"

services:
  front-end:
    build: .
    ports:
      - "8080:80"
    environment:
      - RABBITMQ_USER=${USER}
      - RABBITMQ_PASS=${PASS}
    # uncomment for development
    volumes:
      - "./app:/var/www/html"
  messaging:
    image: rabbitmq
    environment:
      - RABBITMQ_DEFAULT_USER=${USER}
      - RABBITMQ_DEFAULT_PASS=${PASS}
```

Notice how even though the `Dockerfile` copies the app files to `/var/www/html` the `docker-compose.yml` still creates a bind mount over top of them. This allows us to actively develop the PHP code while the container is running. When we are done developing we can comment out the bind mount and the container will use the copied files when the image is rebuilt and run.

You may also notice that `docker-compose.yml` uses some environment variables. It sets them in the container, which we have seen before, *and* it uses the `USER` and `PASS` variables from its own environment. These variables are set in the `.env` file, which Docker Compose reads by default:

front-end-demo/.env

```
USER='front-end-demo'
PASS='changeme'
```

Now that our environment is set up. We can develop our PHP application. We start by augmenting the

example `RpcClient` class to use environment variables for the username and password and `messaging` for the hostname. We will also add an `send` method that sends JSON data.

Within the Docker Compose environment, service names will resolve to the IP addresses of the container within that service. This makes service discovery *much* easier.

Give some thought to what exchange format you want to use. Try to use something that both **Front End** and **Back End** can support. JSON is currently the most common choice.

front-end-demo/app/rpc_client.php

```
<?php

require_once __DIR__ . '/vendor/autoload.php';
use PhpAmqpLib\Connection\AMQPStreamConnection;
use PhpAmqpLib\Message\AMQPMessage;

class RpcClient
{
    private $connection;
    private $channel;
    private $callback_queue;
    private $response;
    private $corr_id;

    public $timeout = 10;
    public $hostname = 'messaging';

    public function __construct()
    {
        $user = $_ENV['MSG_USER'];
        $pass = $_ENV['MSG_PASS'];

        $this->connection = new AMQPStreamConnection(
            $this->hostname,
            5672,
            $user,
            $pass
        );
        $this->channel = $this->connection->channel();
        list($this->callback_queue, ,) = $this->channel->queue_declare(
            "",
            false,
            false,
            true,
```

```
            false
        );
        $this->channel->basic_consume(
            $this->callback_queue,
            '',
            false,
            true,
            false,
            false,
            array(
                $this,
                'onResponse'
            )
        );
    }

    public function onResponse($rep)
    {
        if ($rep->get('correlation_id') == $this->corr_id) {
            $this->response = json_decode($rep->body);
        }
    }

    public function send($action, $data)
    {
        $this->response = null;
        $this->corr_id = uniqid();
        $json = json_encode(array('action'=>$action, 'data'=>$data));

        $msg = new AMQPMessage(
            $json,
            array(
                'correlation_id' => $this->corr_id,
                'reply_to' => $this->callback_queue
            )
        );
        $this->channel->basic_publish($msg, '', 'requests');
        while (!$this->response) {
            $this->channel->wait(null, false, $this->timeout);
        }
        return $this->response;
    }
}
```

To test out our `RpcClient` class, let's create a `action.php` endpoint that takes a POST parameters of `action` and `data` and calls the send procedure with those parameters:

front-end-demo/app/action.php

```
<?php

require('rpc_client.php');

$action = $_POST['action'];
$data = $_POST['data'];

echo "Establishing connection to messaging service... ";
$rpc = new RpcClient();
echo "[SUCCESS]<br>";

echo "Sending {action: \"$action\", data: \"$data\"} to the send procedure and waiting
for response...";
$response = $rpc->send($action, $data);
echo "[SUCCESS]<br>";
echo "Response: <br>";
echo "<pre>";
print_r($response);
echo "</pre>";
```

We'll also need a basic form to test it out:

front-end-demo/app/action.html

```
<html>
    <form action="/action.php" method="post">
        <label for="action">Action:</label>
        <input type="text" name="action">
        <label for="data">Data:</label>
        <input type="text" name="data">
        <input type="submit" value="Submit">
    </form>
</html>
```

Try to avoid using GET parameters for these kinds of endpoints. It makes it too easy for a third party to give a user (presumably authenticated) an evil link that they can just click on and take an action, ie: http://localhost:8080/action.php?action=DO+BAD+THINGS.

Now with these components we should be able to run `docker-compose up` in our `front-end-demo` directory, watch the services start up, and then open up http://localhost:8080/action.html to test the functionality:

```
PS front-end-demo> docker-compose up
Starting front-end-demo_web_1       ... done
Starting front-end-demo_messaging_1 ... done
Attaching to front-end-demo_messaging_1, front-end-demo_web_1
web_1        | AH00558: apache2: Could not reliably determine the server's fully
qualified domain name, using 172.19.0.3. Set the 'ServerName' directive globally to
suppress this message
web_1        | AH00558: apache2: Could not reliably determine the server's fully
qualified domain name, using 172.19.0.3. Set the 'ServerName' directive globally to
suppress this message
web_1        | [Thu May 14 01:24:32.120352 2020] [mpm_prefork:notice] [pid 1] AH00163:
Apache/2.4.38 (Debian) PHP/7.4.5 configured -- resuming normal operations
web_1        | [Thu May 14 01:24:32.120940 2020] [core:notice] [pid 1] AH00094: Command
line: 'apache2 -D FOREGROUND'
messaging_1  | 2020-05-14 01:24:37.668 [info] <0.9.0> Feature flags: list of feature
flags found:
<snip>
messaging_1  | 2020-05-14 01:24:38.252 [info] <0.9.0> Server startup complete; 0 plugins
started.
messaging_1  |  completed with 0 plugins.
web_1        | 172.19.0.1 - - [14/May/2020:01:36:09 +0000] "GET /action.html HTTP/1.1"
200 484 "-" "Mozilla/5.0 (Windows NT 10.0; Win64; x64; rv:76.0) Gecko/20100101
Firefox/76.0" ①
web_1        | 172.19.0.1 - - [14/May/2020:01:36:09 +0000] "GET /favicon.ico HTTP/1.1"
404 489 "-" "Mozilla/5.0 (Windows NT 10.0; Win64; x64; rv:76.0) Gecko/20100101
Firefox/76.0"
messaging_1  | 2020-05-14 01:36:13.729 [info] <0.1183.0> accepting AMQP connection
<0.1183.0> (172.19.0.3:60104 -> 172.19.0.2:5672) ②
messaging_1  | 2020-05-14 01:36:13.731 [info] <0.1183.0> connection <0.1183.0>
(172.19.0.3:60104 -> 172.19.0.2:5672): user ''front-end-demo'' authenticated and granted
access to vhost '/'
messaging_1  | 2020-05-14 01:36:23.756 [warning] <0.1183.0> closing AMQP connection
<0.1183.0> (172.19.0.3:60104 -> 172.19.0.2:5672, vhost: '/', user: ''front-end-demo''):
messaging_1  | client unexpectedly closed TCP connection ③
web_1        | 172.19.0.1 - - [14/May/2020:01:36:13 +0000] "POST /action.php HTTP/1.1"
200 718 "http://localhost:8080/echo.html" "Mozilla/5.0 (Windows NT 10.0; Win64; x64;
rv:76.0) Gecko/20100101 Firefox/76.0" ④
```

① The front-end service will report the pages being served as you are testing it.

② The messaging service will show you the PHP code connecting.

③ At some point we should make `RpcClient` shut down gracefully.

④ Because of how logging is handled, you may not see log messages in the order you expect.

If you test it, this example used the ECHO action and the text "This is a test", you should see the following output after 10 seconds:

```
Establishing connection to messaging service... [SUCCESS] ①
Sending {action: "ECHO", data: "This is a test"} to the send procedure and waiting for
response...[SUCCESS]
Fatal error: Uncaught PhpAmqpLib\Exception\AMQPTimeoutException: The connection timed out
after 10 sec while awaiting incoming data in /var/www/html/vendor/php-amqplib/php-
amqplib/PhpAmqpLib/Wire/AMQPReader.php:141 Stack trace: #0 /var/www/html/vendor/php-
amqplib/php-amqplib/PhpAmqpLib/Wire/AMQPReader.php(163): PhpAmqpLib\Wire\AMQPReader-
>wait() #1 /var/www/html/vendor/php-amqplib/php-
amqplib/PhpAmqpLib/Wire/AMQPReader.php(106): PhpAmqpLib\Wire\AMQPReader->rawread(7) #2
/var/www/html/vendor/php-amqplib/php-
amqplib/PhpAmqpLib/Connection/AbstractConnection.php(566): PhpAmqpLib\Wire\AMQPReader-
>read(7) #3 /var/www/html/vendor/php-amqplib/php-
amqplib/PhpAmqpLib/Connection/AbstractConnection.php(623):
PhpAmqpLib\Connection\AbstractConnection->wait_frame(10) #4 /var/www/html/vendor/php-
amqplib/php-amqplib/PhpAmqpLib/Channel/AbstractChannel.php(234):
PhpAmqpLib\Connection\AbstractConnection->wait_channel(1, 10) #5
/var/www/html/vendor/php-amqplib/php-amqplib/PhpAmqpLib/Channel/AbstractChannel.php(352):
PhpAmqpLib\Channel\Abstrac in /var/www/html/vendor/php-amqplib/php-
amqplib/PhpAmqpLib/Wire/AMQPReader.php on line 141 ②
```

① We can connect to messaging, so we know that works

② But there is nothing listening and responding to our RPC request… yet.

7.3. Resources

7.3.1. Python

- Dockerize Your Flask Application
- Flask Web Framework
- Python: RabbitMQ Tutorial 6 - Remote Procedure Call

7.3.2. JavaScript

- node - Docker Hub
- Node: RabbitMQ Tutorial 6 - Remote Procedure Call

7.3.3. PHP

- php - Docker Hub
- Easy installation of PHP extensions in official PHP Docker images
- composer - Docker Hub
- PHP: RabbitMQ Tutorial 6 - Remote Procedure Call

7.4. Questions

1. *Why are we using a messaging layer for communications with **Front End**?*

2. *What does RPC stand for and what is it?*

3. *What are the most important criteria when picking a language to develop a front end in?*

4. *Describe the most common package management option for one of the following environments: Python, PHP, or JavaScript.*

5. *How are environment variables used in Docker Compose and what kinds of things can they be used for?*

Chapter 8. Back End

8.1. Introduction

Back End is responsible for reading messages from **Front End**, storing things in **Database**, and acquiring any other data needed. It has no user-facing component, which spares us the trouble of having to run another web server. Everything can basically be done with a single script.

8.2. Example

For this example we will be implementing a script in JavaScript that reads and replies to messages via **Messaging**, reads and writes from / to **Database**, and performs some web scraping. JavaScript is being used in contrast to our previous **Front End** coding in PHP to highlight one of the benefits of having a **Messaging** component: we can use different programming languages but still share a common interface. This code is based off of the RabbitMQ Javascript RPC Tutorial.

As stated in Front End, one of the most important parts of being able to work with a programming language is knowing its how it handles package mangement. For node (server-side Javascript), that's the Node Package Manager (NPM). NPM utilizes a `package.json` file that tells it what dependencies to install and how to run the application:

back-end-demo/app/package.json

```
{
  "name": "back-end",
  "version": "1.0.0",
  "description": "Example back end for Systems Integration",
  "main": "server.js",
  "start": "server.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "author": "Ryan Tolboom",
  "license": "ISC",
  "dependencies": {
    "amqplib": "^0.5.6",
    "mariadb": "^2.3.1"
  }
}
```

As you can see, we will be using amqplib to connect to **Messaging** and mariadb to connect to **Database**.

It will also make our lives simpler if we create a custom Docker image that uses npm to install the packages we need *before* our app is run. We will base our's off of the official node image:

back-end-demo/Dockerfile

```
FROM node:12.16.2
COPY ./app/ /home/node/app
WORKDIR /home/node/app
RUN npm install
CMD sh -c 'sleep 10; npm start'
```

If **Back End** comes up before **Messaging** it will be unable to connect. This can be handled in the code for the component, via a separate script, or simply by delaying the start of the component as shown on the CMD line above.

Finally, we will need a `docker-compose.yml` that brings up **Messaging**, **Database**, **Front End**, *and* **Back End**, since **Back End** is the only component that actually uses all three. You'll notice we try not to repeat ourselves by using images that have already been created in the other demos. As we did previously, we will be using environment variables to store some configuration data. These variables can be found in `.env`.

back-end-demo/.env

```
MSG_USER=back-end-demo
MSG_PASS=changeme
DB_ROOT_PASS=changemetoo
DB_USER=car_user
DB_PASS=changemethree
DB_DATABASE=cars
```

back-end-demo/docker-compose.yml

```
version: "3"

services:
  front-end:
    build: ../front-end-demo
    ports:
      - "8080:80"
    environment:
      - MSG_USER=${MSG_USER}
      - MSG_PASS=${MSG_PASS}
  messaging:
    image: rabbitmq:3.8.3
    environment:
      - RABBITMQ_DEFAULT_USER=${MSG_USER}
      - RABBITMQ_DEFAULT_PASS=${MSG_PASS}
  db:
    image: mariadb:10.5.3
    environment:
      - MYSQL_USER=${DB_USER}
      - MYSQL_PASSWORD=${DB_PASS}
      - MYSQL_ROOT_PASSWORD=${DB_ROOT_PASS}
      - MYSQL_DATABASE=${DB_DATABASE}
    volumes:
      - "../db-demo:/docker-entrypoint-initdb.d"
  back-end:
    build: .
    environment:
      - NODE_ENV=development
      - MSG_USER=${MSG_USER}
      - MSG_PASS=${MSG_PASS}
      - DB_USER=${DB_USER}
      - DB_PASS=${DB_PASS}
      - DB_DATABASE=${DB_DATABASE}
```

We use different (albiet bad) passwords for different services. Even though no one outside our Docker network should be able to access these services this is still a good idea.

Our **Back End** script is a basic dispatcher that listens for commands in the **Messaging** `requests` queue, performs actions, and responds via each message's exclusive reply-to queue:

back-end-demo/app/server.js

```
#!/usr/bin/env node
```

```
const https = require('https')

// Messaging configuration
const messaging_host = 'messaging';
const messaging_user = process.env.MSG_USER;
const messaging_pass = process.env.MSG_PASS;
const queue = 'requests';
const messaging_url = `amqp://${messaging_user}:${messaging_pass}@${messaging_host}`;
const amqp = require('amqplib/callback_api');

// Database configuration
const db_host = 'db';
const db_user = process.env.DB_USER;
const db_pass = process.env.DB_PASS;
const db_database = process.env.DB_DATABASE;
const mariadb = require('mariadb/callback');
const db_pool = mariadb.createPool({host: db_host, user: db_user,
  password: db_pass, database: db_database});

function send_response(channel, msg, response) {
  channel.sendToQueue(msg.properties.replyTo, Buffer.from(JSON.stringify(response)), {
    correlationId: msg.properties.correlationId
  });
  channel.ack(msg);
}

amqp.connect(messaging_url, function(error0, connection) {
  if (error0) {
    throw error0;
  }
  connection.createChannel(function(error1, channel) {
    if (error1) {
      throw error1;
    }
    channel.assertQueue(queue, {
      durable: false
    });
    channel.prefetch(1);
    console.log('Listening for requests...');
    channel.consume(queue, function reply(msg) {
      console.log(`Received: ${msg.content}`);
      var received_json = JSON.parse(msg.content);
      var action = received_json['action'];
      switch(action) {
        case 'ECHO':
          data = received_json['data'];
          console.log(`Echoing ${data}`);
```

```
          send_response(channel, msg, {'status': 'OK', 'data': data});
          break;
        case 'CARS':
          console.log('Returning a list of cars from the database');
          db_pool.getConnection((err, conn) => {
            if (err) {
              send_response(channel, msg, {'status': 'ERROR',
                'message': `Unable to connect to database: ${err}`});
            } else {
              conn.query("SELECT * FROM car", (err, rows) => {
                if (err) {
                  send_response(channel, msg, {'status': 'ERROR',
                    'message': `Unable to execute query: ${err}`});
                } else {
                  send_response(channel, msg, {'status': 'OK', 'data': rows});
                }
              });
              conn.end();
            }
          });
          break;
        case 'SCRAPE':
          var date = received_json['data'];
          console.log(`Returning a list of events from NJ.com for ${date}`);
          https.get(
            'https://www.nj.com/web/gateway.php?' +
            'affil=nj&site=default&hidemap=1&tpl=v3_regular_event_grid&' +
            `date=${date}`,
            (res) => {
              res.setEncoding('utf8');
              res.on('data', (body) => {
                var json_data = JSON.parse(body);
                send_response(channel, msg, json_data);
              });
              res.on('error', (error) => {
                send_response(channel, msg, error);
              });
            });
          break;
        default:
          console.log('Unknown action');
          send_response(channel, msg, {'status': 'ERROR', 'message': 'Unknown action'});
          break;
      }
    });
  });
});
```

You almost certainly want to cache replies that are network or computationally intensive. The `SCRAPE` action is a good example of something that could benefit from caching. Do you really need to reach out to NJ.com to get the daily events *every* time? Perhaps they're only updated every 24 hours.

Web scraping is notoriously difficult to keep up with. What happens if NJ.com changes their endpoints? What happens if they detect excessive traffic and decide to throttle your connections?

As you can see the three actions it current supports are:

ECHO reply with whatever data was sent

CARS get list of of cars from our database

SCRAPE get a list of events from NJ.com for a date

While these actions are not particularly useful as currently configured, they do demonstrate the tasks that **Back End** needs to perform: reading / writing from **Messaging**, reading / writing from **Database**, and obtaining additional information from websites.

You should be able to run `docker-compose up` in the `back-end-demo` and test actions with the http://localhost:8080/action.html form. Try the following:

- Action: `ECHO` Data: `This is a test`
- Action: `CARS` Data: ``
- Action: `SCRAPE` Data: `2020-05-18`

Take note of the date format, it is the ISO 8601 standard, very common in web development. Also note that JSON is used as the exchange format, which can be converted easily into native objects on **Front End** and **Back End**.

8.3. Resources

- node - Docker Hub
- docker-node/README - How to use this image
- Beginners Guide to Node Package Manager
- MariaDB Callback API Documentation
- Making HTTP Requests with Node.js

8.4. Questions

1. *What is the common package manager for Node.js?*

2. *What three things does **Back End** have to do?*

3. *How do **Back End** and **Front End** communicate?*

4. *What is an exchange format and why is it important?*

5. *In this example, **Back End** is running in an event loop. What main event does it respond to?*

Chapter 9. WebSockets

9.1. Introduction

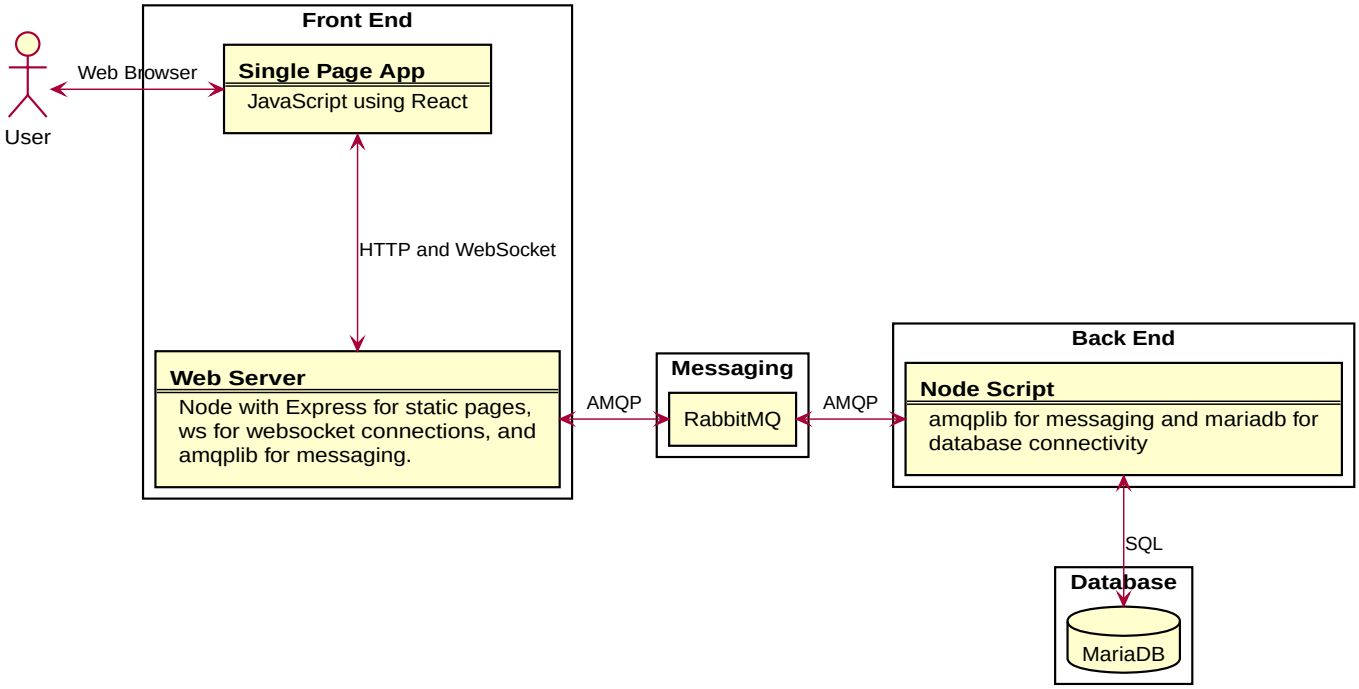
As web apps continue to replace native apps the issues associated with building stateful applications on a platform designed to be stateless often plague developers. Frameworks like React have sprung up to make creating interactive, single-page applications (SPA) easier. These applications tend to more closely mimic their native forebears.

Asynchronous Javascript and XML (AJAX) calls have traditionally been employed to support SPAs, but even with HTTP Keep-Alive employed, these can suffer from connection inefficiencies. Recall our primitive **Front End** PHP example which reconnects to **Messaging** *every* time a page is loaded.

Fortunately a better solution has emerged: WebSockets. WebSockets allow full duplex communication between a client and server over a single HTTP connection. By using WebSockets, we can build an efficient SPA that connects to **Messaging** in the same way as our other **Front End** examples.

9.2. Architecture

This Chapter will focus on the creation of a React **Front End** that utilizes WebSockets. Previously built components will be used for **Messaging**, **Back End**, and **Database**:



9.3. Node Server

Our Node server needs to do two things: serve our React app (which is static JS) and proxy messages from a WebSocket connection to an AMQP connection with **Messaging**. We will use amqplib, Express, and ws to make this possible:

websocket-demo/packages.json

```
{
    "name": "websocket_demo",
    "version": "1.0.0",
    "description": "Using React with RabbitMQ via WebSockets",
    "author": "Ryan Tolboom <ryan.tolboom@njit.edu>",
    "main": "server.js",
    "scripts": {
        "start": "node server.js"
    },
    "dependencies": {
        "express": "^4.17.1",
        "ws": "^7.3.1",
        "amqplib": "^0.6.0"
    }
}
```

Now let's take a look at the source code for our Node server:

websocket-demo/server.js

```
'use strict';

// Messaging configuration
const amqp = require('amqplib');
const messaging_host = 'messaging';
const messaging_user = process.env.MSG_USER;
const messaging_pass = process.env.MSG_PASS;
const queue = 'requests';
const messaging_url = `amqp://${messaging_user}:${messaging_pass}@${messaging_host}`;
// use one connection for all AMPQ things
var messaging_conn = amqp.connect(messaging_url);

// Express configuration
const express = require('express');
const app = express();
const port = 8080;
// serve static files from the current dir
app.use(express.static(__dirname))
const server = app.listen(port);

// ws configuration
const ws = require('ws');
const wsServer = new ws.Server({ noServer: true });

// since we are sharing a server we pass off upgrade requests to ws
server.on('upgrade', (request, socket, head) => {
    wsServer.handleUpgrade(request, socket, head, socket => {
        wsServer.emit('connection', socket, request);
    });
});

// Set up a WebSocket AMPQ proxy
wsServer.on('connection', function(socket) {
    console.log("WebSocket connection established");
    messaging_conn.then(function(conn) {
        console.log("Building AMQP channel");
        conn.createChannel().then(function(ch) {
            console.log("Creating AMPQ response queue");
            ch.assertQueue('', { exclusive: true }).then(function(ok) {
                console.log("Creating callback for AMPQ response messages");
                ch.consume(ok.queue, function(response) {
                    console.log(`AMQP ${ok.queue} -> WebSocket`);
                    socket.send(response.content.toString());
                }, { noAck: true });
                return ok;
            }).then(function(ok) {
                console.log("Creating a callback for WebSocket messages");
                socket.on('message', function(message) {
```

```
                    console.log(`WebSocket -> AMQP ${queue}`);
                    ch.sendToQueue(queue, Buffer.from(message),
                        { replyTo: ok.queue });
                });
            });
        });
    });
});

console.log("Serving static files and listening for WebSocket connections...");
```

The ampqlib configuration has been covered already in Back End. The Express configuration is relatively simple, specifying a static directory to serve and a port to listen on. In this case we do not even need to define endpoints as we would in a Flask application.

The ws configuration is where the bulk of our code is. First we have to specify that when a client signals that it wants to upgrade an HTTP connection Express passes control over to the wsServer we created. Secondly we have establish a method for routing messages from the WebSocket to **Messaging** and vice versa. The following actions are taken *when a WebSocket is established*:

1. A new AMPQ channel is opened over an existing connection to RabbitMQ. By having a standing connection to **Messaging** we reduce the overhead.
2. A new, exclusive queue is created to receive messages from **Messaging**. Each client has their own receive queue to make sure messages are not mixed up.
3. A callback is established to consume messages on the receive queue and forward them to the WebSocket.
4. A callback is established to take messages from the WebSocket and forward them to the requests queue.

Callbacks/Promises/Await: You may notice that Express and ws rely heavily on the use of callbacks, but ampqlib looks slightly different with its `then()` chains. Another syntax you may see is the use of the `await` keyword in `async` functions, waiting for the resolution of Promises. Any way you slice it, you will be working with *asynchronous code*, namely situations where you don't want to be busy waiting for something to resolve when you could be working on something else. Get used to these situations and carefully read the documentation for the libraries you are using to make your code more efficient.

Why aren't we using TypeScript? TypeScript is a great way to fix many of the problems the vanilla JS suffers from. The reason this example does not use TypeScript is to keep things as simple as possible. If you're working on a larger project TypeScript and Node go great together.

Keep the security implications in mind when creating these types of tunnels. We have effectively given anyone direct access to **Messaging**. Is this something we are prepared for?

9.4. React Client

Let's discuss the code that or Node server is actually serving. In the `websocket-demo` directory you will find a very simple `index.html` that is presented as the default page when visiting http://localhost:8080:

websocket-demo/index.html

```
<html lang="en">
<head>
    <meta charset="utf-8">
    <title>React Front End Demo</title>
</head>
<body>
    </html>
        <h1>React Front End Demo</h1>
        <div id="root"></div>
        <script src="https://unpkg.com/react@16/umd/react.development.js"
crossorigin></script>
        <script src="https://unpkg.com/react-dom@16/umd/react-dom.development.js"
crossorigin></script>
        <script src="app.js"></script>
    </html>
</body>
```

All this page does is load React, provide a `#root` div for our app, and load our React app (app.js):

websocket-demo/app.js

```
'use strict';

const e = React.createElement;

class Messages extends React.Component {
    messageEvent(event) {
        this.setState({messages: this.state.messages.concat(event.data)});
    }

    constructor(props) {
        super(props);
        this.state = {messages: []};
        props.socket.addEventListener('message', this.messageEvent.bind(this));
    }
```

```
    render() {
        return this.state.messages.map(
            (message) => e('div', null, `${message}`)
        )
    }
}

class Send extends React.Component {
    constructor(props) {
        super(props);
        this.state = {action: '', data: ''};
        this.changeAction = this.changeAction.bind(this);
        this.changeData = this.changeData.bind(this);
        this.handleSubmit = this.handleSubmit.bind(this);
    }

    changeAction(event) {
        this.setState({action: event.target.value, data: this.state.data});
    }

    changeData(event) {
        this.setState({action: this.state.action, data: event.target.value});
    }

    handleSubmit(event) {
        this.props.socket.send(
            JSON.stringify({action: this.state.action, data: this.state.data}));
        this.setState({action: '', data: ''});
        event.preventDefault();
    }

    render() {
        return e("form", {onSubmit: this.handleSubmit},
                e("label", null, "Action:",
                    e("input", {type: "text", value: this.state.action,
                        onChange: this.changeAction})),
                e("label", null, "Data:",
                    e("input", {type: "text", value: this.state.data,
                        onChange: this.changeData})),
                e("input", {type: "submit", value: "Send"}));
    }
}

class App extends React.Component {
    constructor(props) {
        super(props);
        this.state = {socket: new WebSocket('ws://localhost:8080')}
```

```
    }

    render() {
        return [
            e(Messages, {socket: this.state.socket}),
            e(Send, {socket: this.state.socket})
        ]
    }
}

ReactDOM.render(e(App), document.getElementById('root'));
```

The React app is made up of two components: `Messages` and `Send`. `Messages` keeps a list of received messages, renders them as div elements, and provides a callback for adding new messages. `Send` renders a form with two text inputs, tracks their values, and creates a JSON message when the submit button is pressed.

A WebSocket connection is created through the WebSocket API in the constructor of our main component: `App`. This socket is passed to `Send` and `Messages` which use it to send and listen for messages respectively.

Why no JSX? JSX is a very interesting tool for legibly putting HTML inside JS. Given how little HTML there actually is in this example, it was not used to keep things simple. If you are interested in using it, I would recommend giving it a try. As apps become increasingly complex it can greatly simplify the look of your code.

9.5. Docker Environment

Our environment variables should look familiar, we used them before in the previous demos. A user name and password for **Messaging** and **Database** as well as which database we would like to use:

websocket-demo/.env

```
MSG_USER=websocket-demo
MSG_PASS=changeme
DB_ROOT_PASS=changemetoo
DB_USER=car_user
DB_PASS=changemethree
DB_DATABASE=cars
```

We will be building off of the Docker Hub node image, installing the packages we need, copying our app, and introducing a slight delay so that **Messaging** starts up before our container:

websocket-demo/Dockerfile

```
FROM node:14
WORKDIR /app
COPY package.json ./
RUN npm install
COPY . .
EXPOSE 8080
CMD /bin/sh -c "sleep 10; node server.js"
```

Our `websocket-demo/docker-compose.yml` file should look familiar as well, with the exception of the `react-front-end` service it is using the **Messaging**, **Back End**, and **Database** we developed earlier. The `react-front-end` service builds from the `Dockerfile` in the current directory, uses port `8080` (on the host *and* the container), and adds the **Messaging** login information to the container environment:

websocket-demo/docker-compose.yml

```
version: "3"
services:
  react-front-end:
    build: .
    ports:
      - "8080:8080"
    environment:
      - NODE_ENV=development
      - MSG_USER=${MSG_USER}
      - MSG_PASS=${MSG_PASS}
    #uncomment for development
    volumes:
      - ".:/app"
  messaging:
    image: rabbitmq:3.8.9-management
    ports:
      - "15672:15672"
    environment:
      - RABBITMQ_DEFAULT_USER=${MSG_USER}
      - RABBITMQ_DEFAULT_PASS=${MSG_PASS}
  db:
    image: mariadb:10.5.5
    environment:
      - MYSQL_USER=${DB_USER}
      - MYSQL_PASSWORD=${DB_PASS}
      - MYSQL_ROOT_PASSWORD=${DB_PASS}
      - MYSQL_DATABASE=${DB_DATABASE}
    volumes:
      - "../db-demo:/docker-entrypoint-initdb.d"
  back-end:
    build: ../back-end-demo
    environment:
      - NODE_ENV=development
      - MSG_USER=${MSG_USER}
      - MSG_PASS=${MSG_PASS}
      - DB_USER=${DB_USER}
      - DB_PASS=${DB_PASS}
      - DB_DATABASE=${DB_DATABASE}
```

9.6. Testing it Out

Running `docker-compose up` in the `websocket-demo` should bring everything up:

```
PS C:\Users\rxt1077\IT490\websocket-demo> docker-compose up
Starting websocket-demo_react-front-end_1 ... done
Starting websocket-demo_db_1              ... done
Starting websocket-demo_messaging_1       ... done
Starting websocket-demo_back-end_1        ... done
Attaching to websocket-demo_back-end_1, websocket-demo_react-front-end_1, websocket-
demo_db_1, websocket-demo_messaging_1
<snip>
db_1              | 2020-10-03 14:50:57 0 [Note] mysqld: ready for connections.
db_1              | Version: '10.5.5-MariaDB-1:10.5.5+maria~focal'  socket:
'/run/mysqld/mysqld.sock'  port: 3306  mari
adb.org binary distribution
<snip>
messaging_1       | 2020-10-03 14:51:02.440 [info] <0.516.0> Server startup complete; 4
plugins started.
messaging_1       |  * rabbitmq_prometheus
messaging_1       |  * rabbitmq_management
messaging_1       |  * rabbitmq_web_dispatch
messaging_1       |  * rabbitmq_management_agent
messaging_1       |  completed with 4 plugins.
messaging_1       | 2020-10-03 14:51:02.440 [info] <0.516.0> Resetting node maintenance
status
back-end_1        |
back-end_1        | > back-end@1.0.0 start /home/node/app
back-end_1        | > node server.js
back-end_1        |
messaging_1       | 2020-10-03 14:51:05.496 [info] <0.824.0> accepting AMQP connection
<0.824.0> (172.24.0.2:39522 -> 1
72.24.0.5:5672)
messaging_1       | 2020-10-03 14:51:05.555 [info] <0.824.0> connection <0.824.0>
(172.24.0.2:39522 -> 172.24.0.5:5672)
: user 'back-end-demo' authenticated and granted access to vhost '/'
back-end_1        | Listening for requests...
react-front-end_1 | Serving static files and listening for WebSocket connections...
messaging_1       | 2020-10-03 14:51:08.086 [info] <0.839.0> accepting AMQP connection
<0.839.0> (172.24.0.3:37848 -> 1
72.24.0.5:5672)
messaging_1       | 2020-10-03 14:51:08.135 [info] <0.839.0> connection <0.839.0>
(172.24.0.3:37848 -> 172.24.0.5:5672)
: user 'back-end-demo' authenticated and granted access to vhost '/'
```

Trimming down some of the massive startup messages (looking at you **Messaging**) you can see that everything started successfully. Now we can go to http://localhost:8080 and we should see our React client being served. You will notice in the logs that our `react-front-end` Node server builds its channel and callbacks once that WebSocket is established:

```
react-front-end_1  | WebSocket connection established
react-front-end_1  | Building AMQP channel
react-front-end_1  | Creating AMPQ response queue
react-front-end_1  | Creating callback for AMPQ response messages
react-front-end_1  | Creating a callback for WebSocket messages
```

Finally, let's send an `ECHO` action with `Hello World` as our data and confirm that everything is working:

```
react-front-end_1  | WebSocket -> AMQP requests
back-end_1         | Received: {"action":"ECHO","data":"Hello World!"}
back-end_1         | Echoing Hello World!
react-front-end_1  | AMQP amq.gen-vg0n7kAPJSMi4j8NCnYXiA -> WebSocket
```

We also see the response displayed in the app as a new div element. Notice that our request was sent to the `requests` queue and the response was sent back to an exclusive queue we created (`amq.gen-vg0n7kAPJSMi4j8NCnYXiA` in this case).

Using WebSockets for an RPC client does not really leverage the full power of the connection. With WebSockets you can push messages to the connected clients to enable all sorts of interesting things: notifications, broadcast messages, etc. What's shown here is really just the tip of the iceberg.

9.7. Questions

1. *What are the advantages of this front end, as opposed to the front end demonstrated in Front End?*

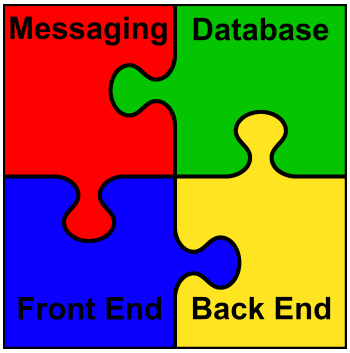
2. *What is a WebSocket and what does it allow you to do?*

3. *What happens when a new WebSocket connection is made?*

4. *What is meant by upgrading a HTTP connection and why do we configure Express to hand off those upgrades to ws?*

5. *Imagine you want the **Back End** to be able to broadcast messages to all connected clients. How would you implement that with this system?*

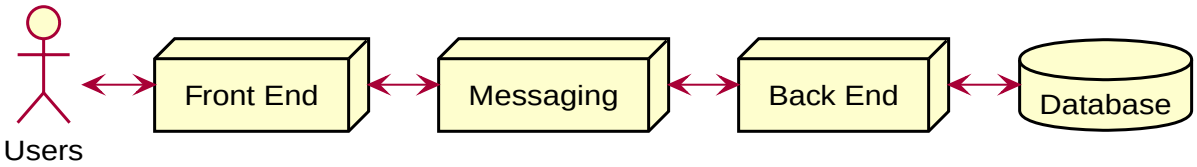
Chapter 10. Midterm Example



10.1. Introduction

This section summarizes the content so far and provides an example of a system that implements basic the midterm milestones. This example is just one way to build a system . It's not the only way, in fact it's not even the best way. Your system will likely be quite different to meet the individual needs of your project. 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.

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.

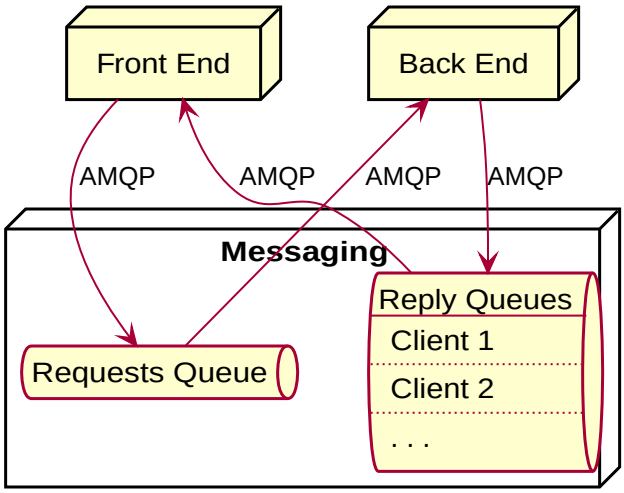
The entire example can be found in the `example-midterm` directory which has the following directory structure:

```
|    .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.

10.2. 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.

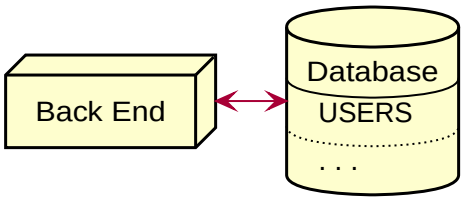
The service definition in `docker-compose.yml` can be seen here:

example-midterm/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
```

10.3. 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 `example-midterm/db/Dockerfile` handles copying our initialization SQL appropriately:

example-midterm/db/Dockerfile

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

example-midterm/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 in the `docker-compose.yml` file can be seen here:

example-midterm/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.

10.4. 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

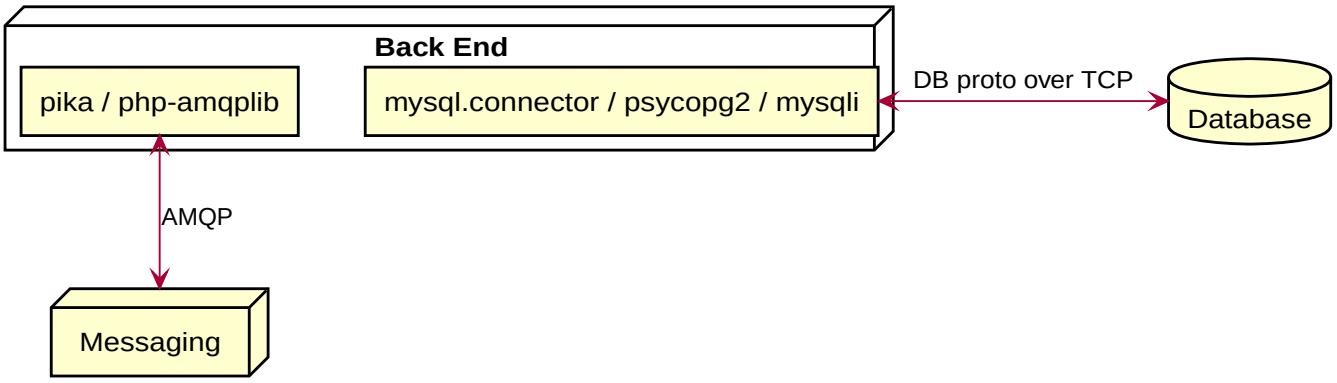
- psycopg2 (PostgreSQL)
- mysql.connector (MySQL / MariaDB)
- pika (RabbitMQ)

PHP

- mysqli (MySQL / MariaDB)
- php-amqplib (RabbitMQ)

JavaScript

- node-postgres (PostgreSQL)
- mariadb (MySQL / MariaDB)
- amqplib (RabbitMQ)



The code for the **Back End** example is entirely contained in `example-midterm/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:

```
PS example-midterm> 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:

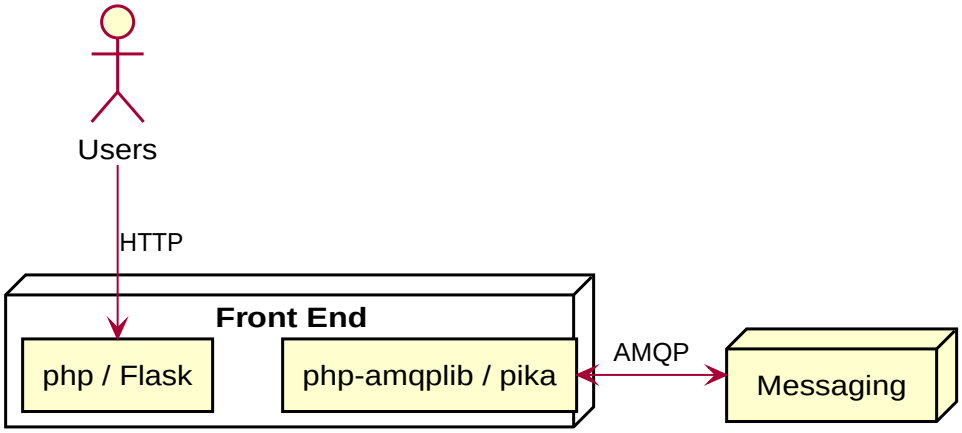
example-midterm/back_end/app.py (excerpted)

```
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.

10.5. Front End

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 `example-midterm/front_end/Dockerfile` is used for creating the image:

example-midterm/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 **Messaging** is up before **Front End** starts. This way **Front End** will not give errors to users who attempt to use it before the full system has started.

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

example-midterm/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 this 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** easier, `example-midterm/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.

example-midterm/front_end/messaging.py

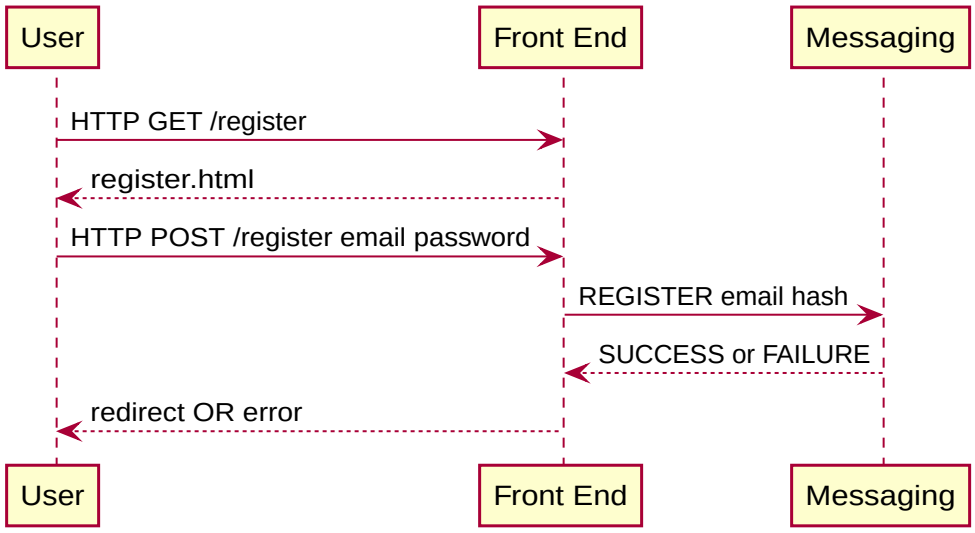
```
import pika
import json
import time
import logging
import os

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 `example-midterm/front_end/app.py` follows:

example-midterm/front_end/app.py (excerpted)

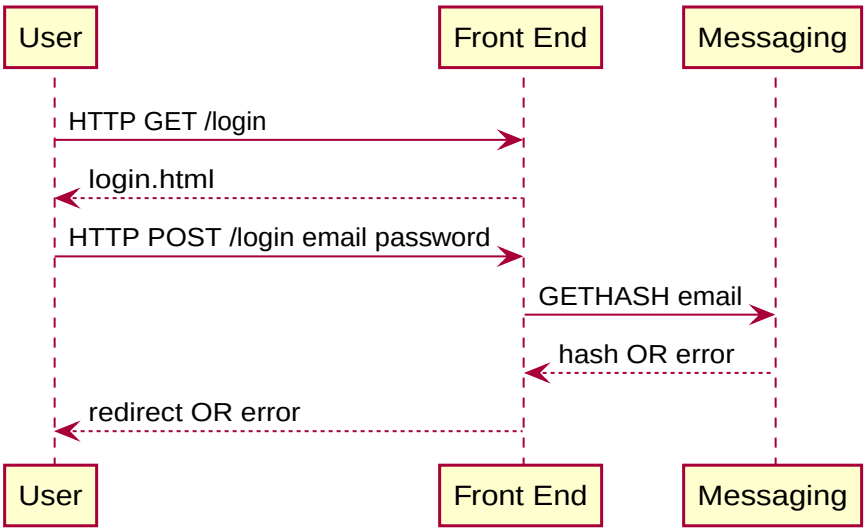
```
@app.route('/register', methods=['GET', 'POST'])
def register():
    if request.method == 'POST':
        email = request.form['email']
        password = request.form['password']
        msg = messaging.Messaging()
        msg.send(
            'REGISTER',
            {
                'email': 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 `example-midterm/front_end/app.py` follows:

example-midterm/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 `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:

example-midterm/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
```

10.6. Questions

1. *What is the advantage of using a `.env` file and referencing it from docker-compose.yml?*

   A `.env` file allows you to have a single source of information for things that may be used by multiple services. This way you only have to make a change in one file if you want to make an adjustment. This helps prevent errors due to forgetting to change a variable in multiple places. It also allows you to store your secrets in one file that you can then keep out of your public repository.

2. *Why do we store password hashes instead of just the passwords?*

   Hashes are more secure because they are a one-way function. You can convert a password to its hash, but you *cannot* convert a hash to its password. As such, if your system is compromised an attacker would not get the passwords of your users.

3. *Why is port `5000` the only port that needs to be forwarded to the local host?*

   All other communication in this system takes place inside the Docker network that Docker Compose builds. The only interface a user needs to this system is through port 5000 on **Front End**.

4. *What does the `adminer` image do?*

   Adminer provides a web-based interface to your database. This can be very useful for looking at the data in your database and making sure everything is working the way you expect it to.

5. *What role does **Messaging** play in this system?*

   Messaging brokers the communication between **Front End** and **Back End**. By using a **Messaging** layer, we make our system more scalable. In the future we may wish to run multiple **Front Ends** or **Back Ends**. With a **Messaging** layer that transition should be simple.

Chapter 11. 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.

11.1. Background

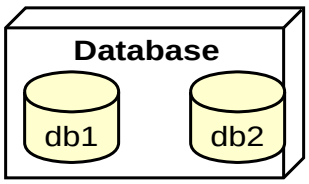


Figure 8. Basic Replication

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:

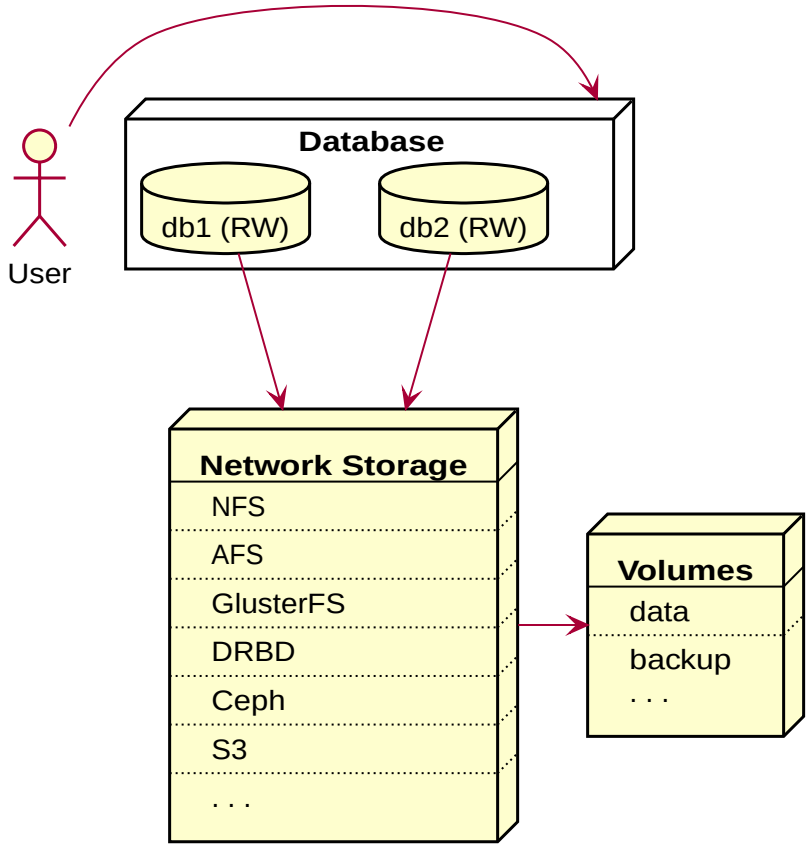


Figure 9. Volume Sharing

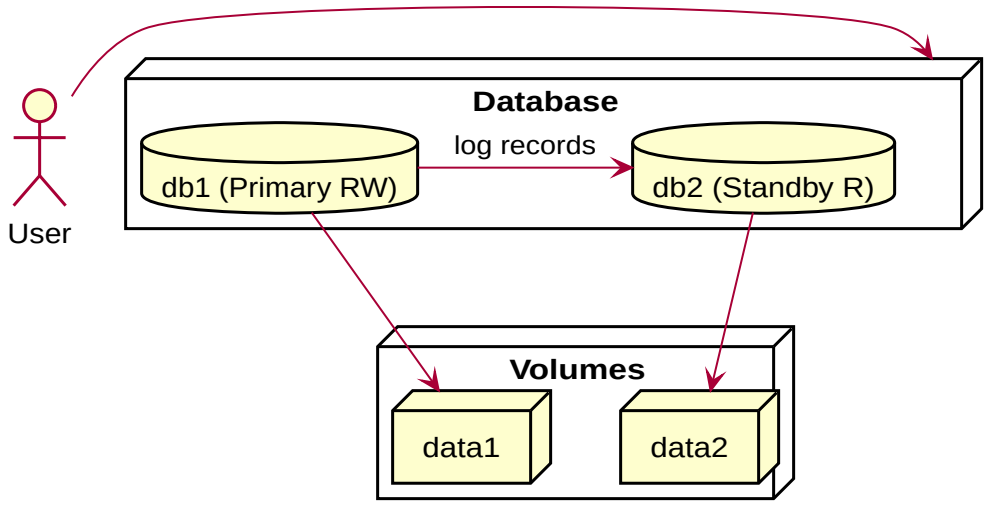


Figure 10. Hot / Warm Standby

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.

11.2. Implementation

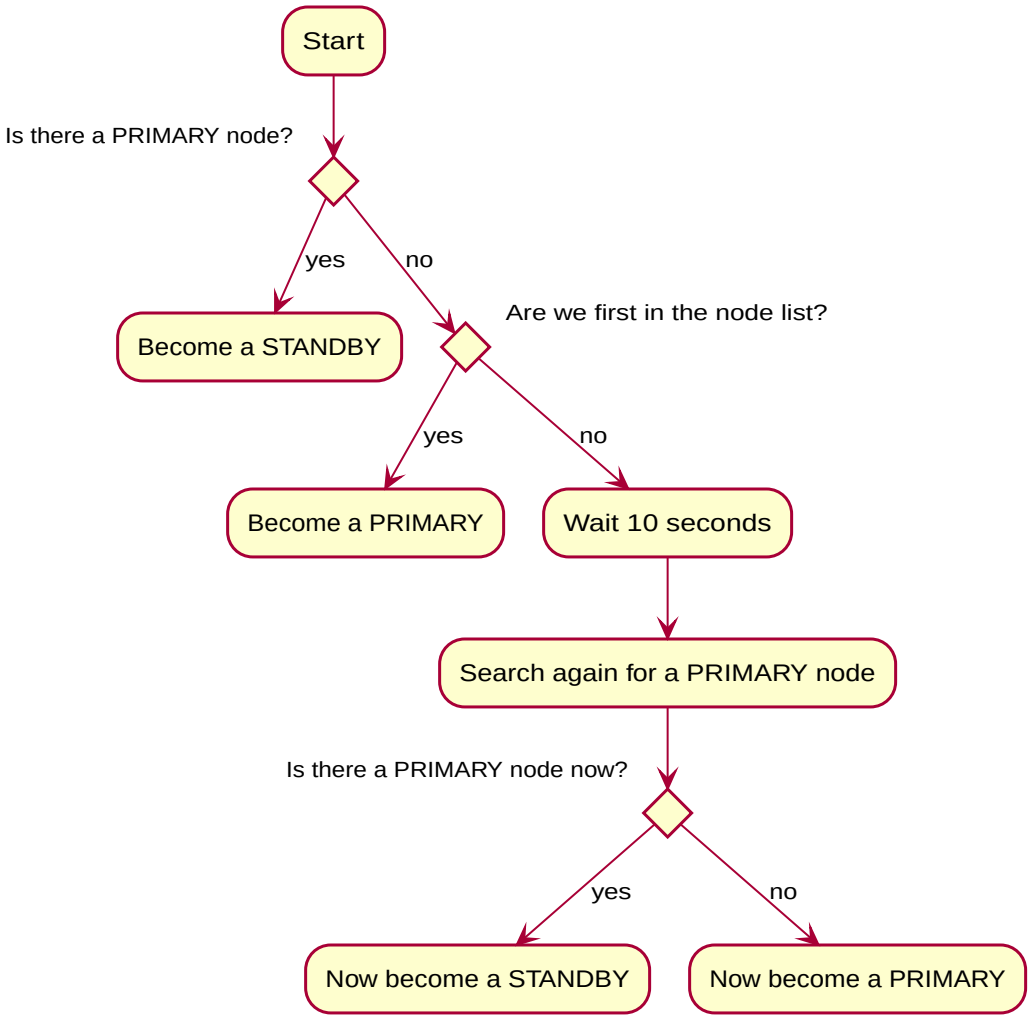
This example implements a Hot Standby in PostgreSQL. Actually implementing *replication* is largely handled for us by the DMBS. It is simply a matter of setting configuration variables, starting the databases are in the correct state, and bringing up the services in the right order. We can handle this 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.

11.3. 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:

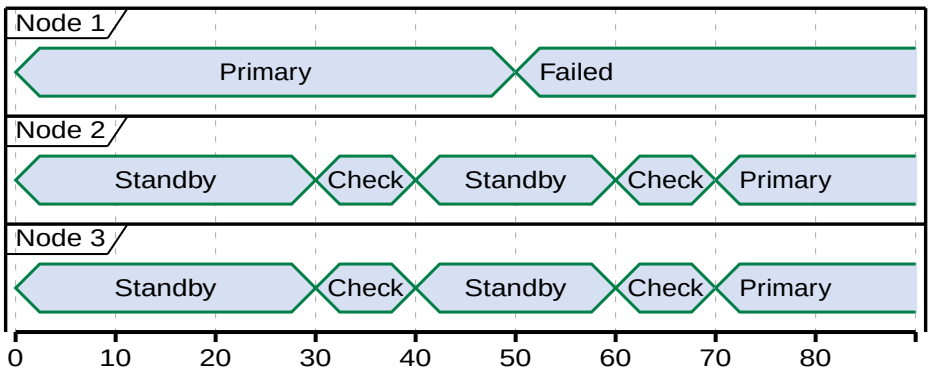


Figure 11. Dual Promotion

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.

11.4. 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.

11.5. 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 12. Kubernetes

12.1. 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:

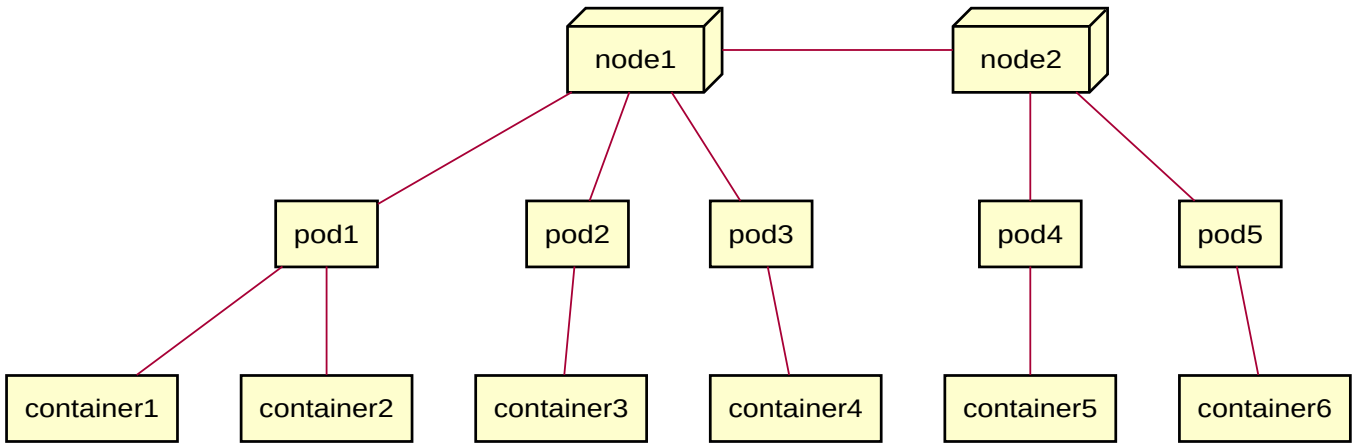


Figure 12. Kubernetes Cluster

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

12.2. 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:

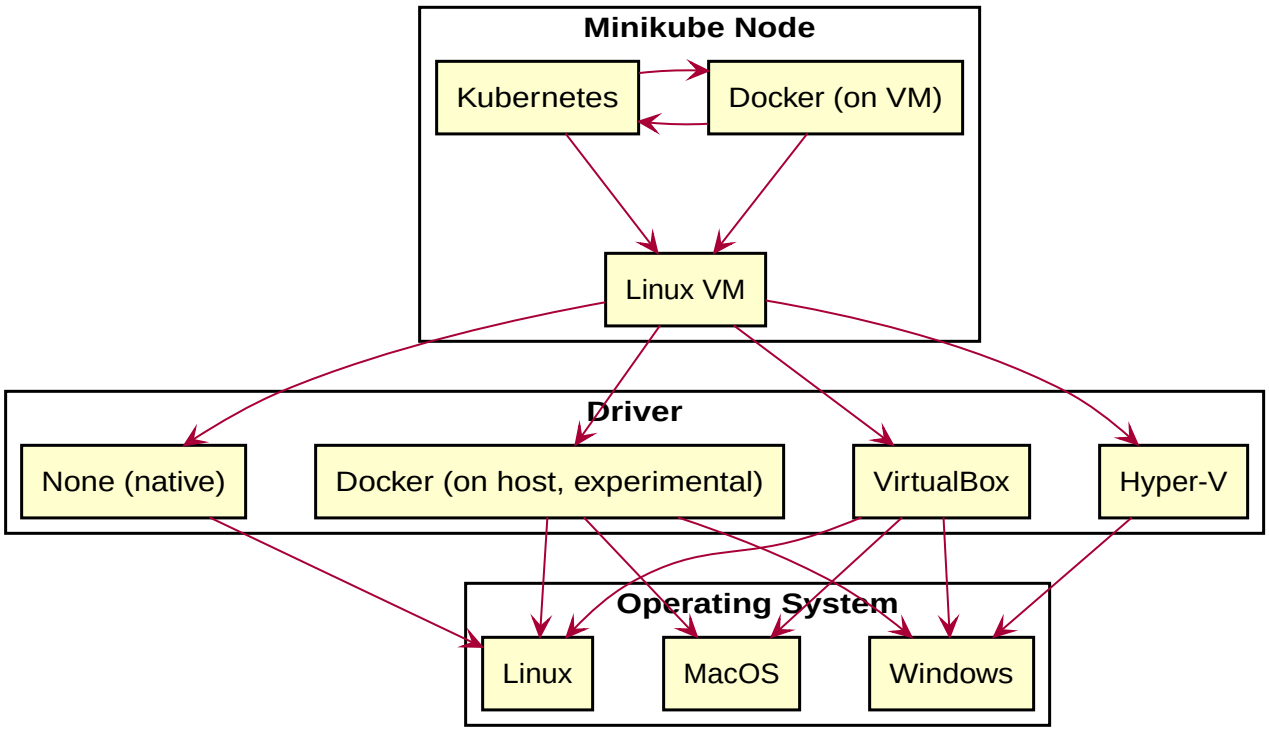


Figure 13. Minikube Architecture

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.
```

12.3. 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/`

12.4. 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.

12.5. Questions

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

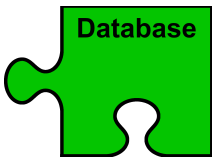
2. *What does a **Deployment** do?*

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 13. Database in Kubernetes



13.1. 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:

13.2. 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

13.3. 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.

13.4. 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.

13.5. 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.

13.6. 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.

13.7. Resources

- Run a Replicated Stateful Application

13.8. 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 14. Messaging in Kubernetes

14.1. Introduction

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

14.2. 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.

14.3. 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.

14.4. Example

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

14.4.1. 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.

14.4.2. 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.

14.4.3. 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).

14.4.4. 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.

14.4.5. 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.local, 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, rabbit@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.local: {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.local: {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 node...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.local, 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, rabbit@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.local: {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-clustering②
<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.local, 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, rabbit@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-clustering
<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-node 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-node 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-node 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.

14.5. 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

14.6. 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?*

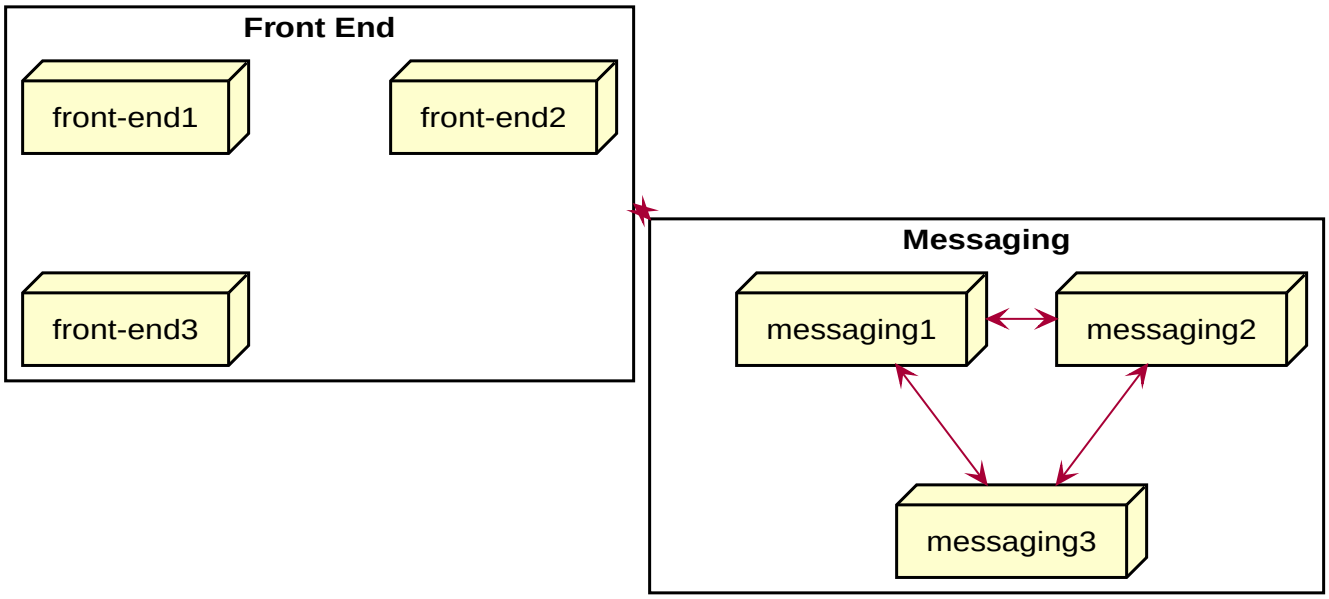
4. *What role does RBAC play in the Kubernetes cluster?*

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

Chapter 15. Front End in Kubernetes

15.1. 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.

15.2. 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.

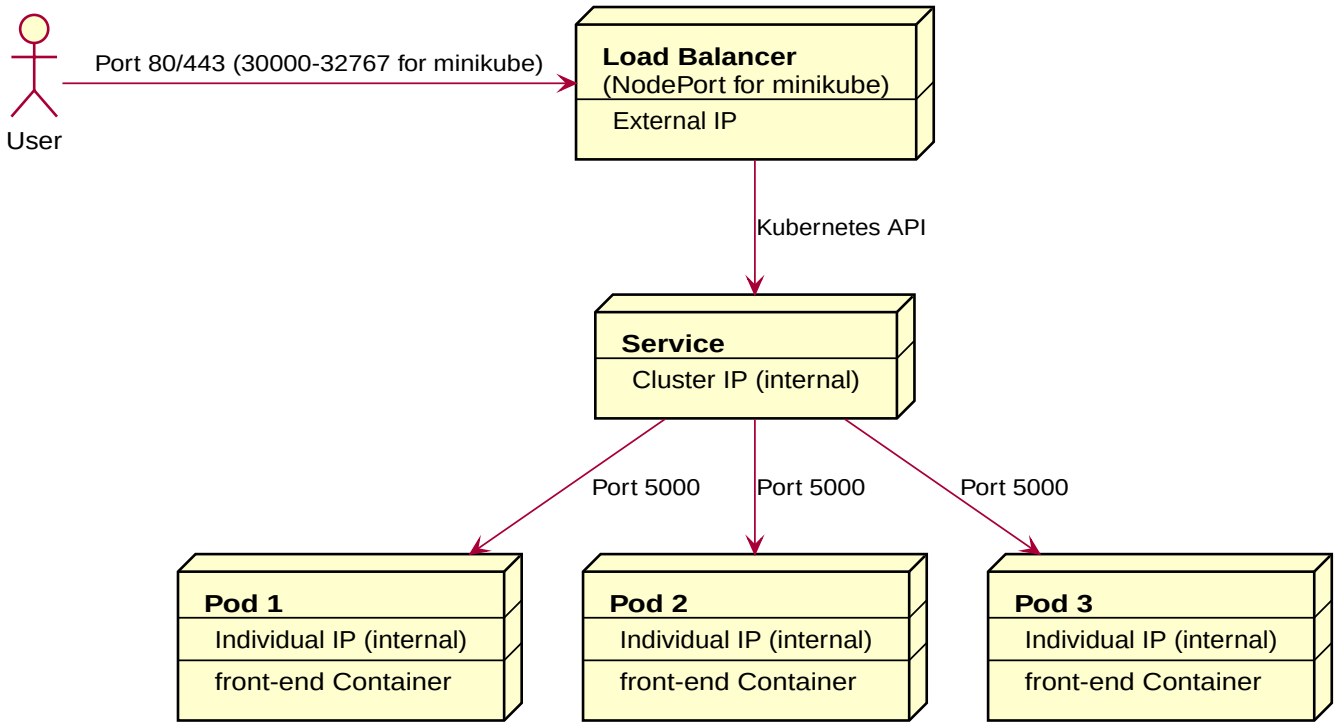


Figure 14. Load Balancer Architecture

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.

15.3. Example

15.3.1. 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.

15.3.2. 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.

15.3.3. 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.

15.3.4. 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
          #image: gcr.io/example-20200503/front-end:v1
          env:
            - name: RABBITMQ_DEFAULT_USER
              value: "guest"
            - name: RABBITMQ_DEFAULT_PASS
              value: "guest"
            - name: FLASK_SECRET_KEY
              value: "changeme"
```

15.3.5. 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.

15.4. 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 16. Back End in Kubernetes

16.1. Introduction

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.

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:

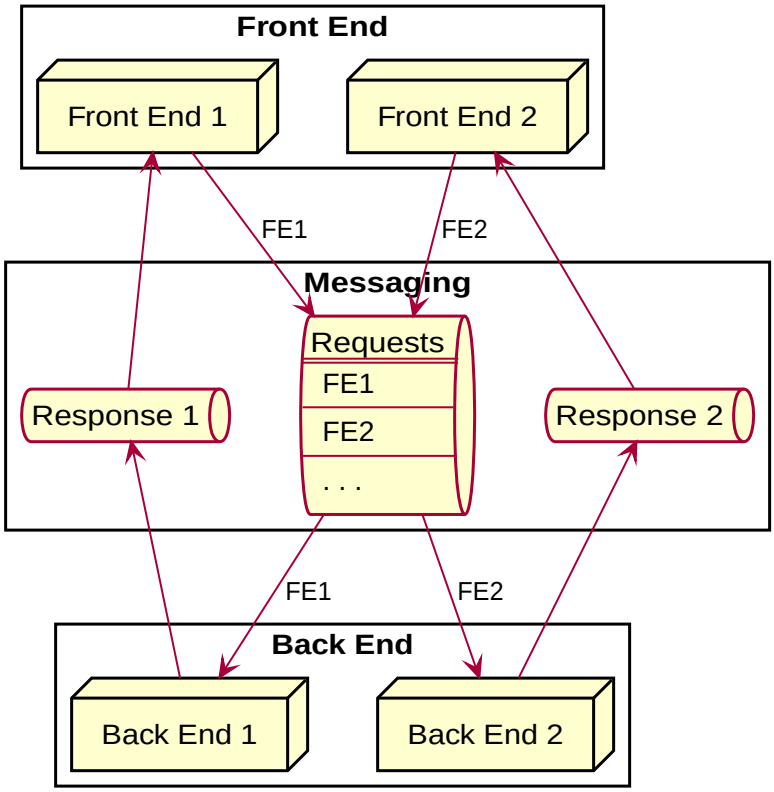


Figure 15. **Back End** *Architecture*

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.

16.2. 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
          #image: gcr.io/example-20200503/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.

16.3. 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?*

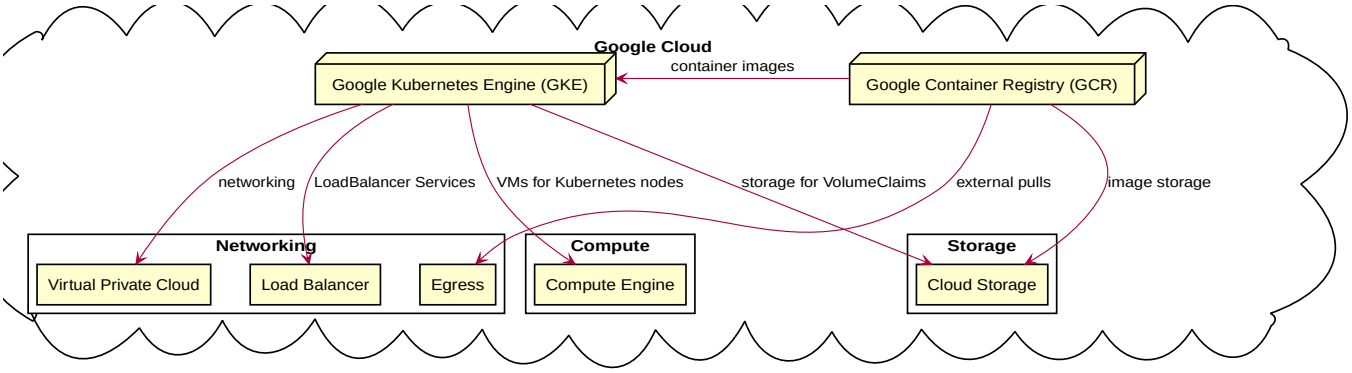
Chapter 17. Google Kubernetes Engine

17.1. 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 (GKE). If you want to follow along with 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 add a credit card to your account and *you can be charged money for the services you use*. If you are worried about incurring an expense, feel free to simply read through the examples.

Google Cloud provides many services that work with each other. We will be using GKE and Google Container Registry (GCR) which in-turn will be using other supporting services:



17.2. Setting up gcloud

We will be doing as much of this is possible from the command line so 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. 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 a timestamp as part of your ID: `example-2020428`.

Now that we have gcloud installed and we have a project, we need to specify what zone/region we would like to use for the project. Google Cloud has many geographic regions with multiple zones available depending on where you are expecting your application to be used and what you're computer needs are, respectively. In this example we will set it to `us-central1-c` with the following command:

```
PS example-final> gcloud config set compute/zone us-central1-c
Updated property [compute/zone].
```

Now let's try to add a Kubernetes cluster named `example` and see what happens:

```
PS example-final> gcloud container clusters create example
WARNING: Currently VPC-native is not the default mode during cluster creation. In the
future, this will become the default mode
and can be disabled using `--no-enable-ip-alias` flag. Use `--[no-]enable-ip-alias` flag
to suppress this warning.
WARNING: Newly created clusters and node-pools will have node auto-upgrade enabled by
default. This can be disabled using the `-
-no-enable-autoupgrade` flag.
WARNING: Starting with version 1.18, clusters will have shielded GKE nodes by default.
WARNING: Your Pod address range (`--cluster-ipv4-cidr`) can accommodate at most 1008
node(s).
This will enable the autorepair feature for nodes. Please see
https://cloud.google.com/kubernetes-engine/docs/node-auto-repair f
or more information on node autorepairs.
ERROR: (gcloud.container.clusters.create) ResponseError: code=403, message=Kubernetes
Engine API is not enabled for this project
. Please ensure it is enabled in Google Cloud Console and try again: visit
https://console.cloud.google.com/apis/api/container.g
oogleapis.com/overview?project=example-20200428 to do so.
```

Follow the directions in the error, visit the URL specified (it will be different depending on your project name), and enable the GKE API for this project.

Make sure you are signed in with the Google account that you linked to Google Cloud. If you want to be certain, you can open an incognito / private browsing tab and paste the URL in there. That will force you to have to log in.

Once we've enabled the GKE API, let's try our `create cluster` command again:

```
PS example-final> gcloud container clusters create example
WARNING: Currently VPC-native is not the default mode during cluster creation. In the
future, this will become the default mode
and can be disabled using `--no-enable-ip-alias` flag. Use `--[no-]enable-ip-alias` flag
to suppress this warning.
WARNING: Newly created clusters and node-pools will have node auto-upgrade enabled by
default. This can be disabled using the `-
-no-enable-autoupgrade` flag.
WARNING: Starting with version 1.18, clusters will have shielded GKE nodes by default.
WARNING: Your Pod address range (`--cluster-ipv4-cidr`) can accommodate at most 1008
node(s).
This will enable the autorepair feature for nodes. Please see
https://cloud.google.com/kubernetes-engine/docs/node-auto-repair f
or more information on node autorepairs.
Creating cluster example in us-central1-c... Cluster is being health-checked (master is
healthy)...done.
Created [https://container.googleapis.com/v1/projects/example-20200428/zones/us-central1-
c/clusters/example].
To inspect the contents of your cluster, go to:
https://console.cloud.google.com/kubernetes/workload_/gcloud/us-central1-c/examp
le?project=example-20200428
kubeconfig entry generated for example.
NAME     LOCATION       MASTER_VERSION  MASTER_IP       MACHINE_TYPE   NODE_VERSION
NUM_NODES  STATUS
example  us-central1-c  1.14.10-gke.27  35.223.164.188  n1-standard-1  1.14.10-gke.27  3
RUNNING
```

Our last step will be setting up Docker to use our gcloud credentials:

```
PS example-final> gcloud auth configure-docker
Adding credentials for all GCR repositories.
WARNING: A long list of credential helpers may cause delays running 'docker build'. We
recommend passing the registry name to co
nfigure only the registry you are using.
After update, the following will be written to your Docker config file
 located at [.docker\config.json]:
 {
  "credHelpers": {
    "gcr.io": "gcloud",
    "marketplace.gcr.io": "gcloud",
    "eu.gcr.io": "gcloud",
    "us.gcr.io": "gcloud",
    "staging-k8s.gcr.io": "gcloud",
    "asia.gcr.io": "gcloud"
  }
}

Do you want to continue (Y/n)?  Y

Docker configuration file updated.
```

Congratulations! You've now downloaded and installed gcloud and built your first cluster in Google Cloud. From here on, we will be working with more familiar utilities: kubectl and Docker.

17.3. Pushing Images

Unlike minikube, which we set up to use images from its native Docker daemon (remember `minikube docker-env`?), GKE expects to be able to pull custom images from an actual repository. Since we're already using Google Cloud, it makes sense to push our custom images to GCR. Here is how we do that:

```
PS example-final> docker build -t gcr.io/example-20200428/front-end:v1 ./front-end/ ①
Sending build context to Docker daemon   16.9kB
Step 1/6 : FROM python:3.9.0a5-buster
3.9.0a5-buster: Pulling from library/python
90fe46dd8199: Pull complete
35a4f1977689: Pull complete
bbc37f14aded: Pull complete
74e27dc593d4: Pull complete
4352dcff7819: Pull complete
deb569b08de6: Pull complete
c7360a3495cf: Pull complete
f58442eaf6a4: Pull complete
617f2eb777a8: Pull complete
Digest: sha256:f7251883daa3d6484055af80ebcd72f083d58de9276ee772d95d2dc50e0ea951
```

```
Status: Downloaded newer image for python:3.9.0a5-buster
 ---> b5f66cb660dd
Step 2/6 : COPY . /app
 ---> f288da00c29c
Step 3/6 : WORKDIR /app
 ---> Running in 40540d7524d3
Removing intermediate container 40540d7524d3
 ---> 1da446e063c3
Step 4/6 : RUN pip install -r requirements.txt
 ---> Running in 0873978faef8
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-7.1.2-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
  Downloading MarkupSafe-1.1.1.tar.gz (19 kB)
Building wheels for collected packages: MarkupSafe
  Building wheel for MarkupSafe (setup.py): started
  Building wheel for MarkupSafe (setup.py): finished with status 'done'
  Created wheel for MarkupSafe: filename=MarkupSafe-1.1.1-cp39-cp39-linux_x86_64.whl
size=32073 sha256=ff3d6994faed1b54ea40c09ef
64f972aec74cb3f3d77fbdc55335143c959bcea
  Stored in directory:
/root/.cache/pip/wheels/e0/19/6f/6ba857621f50dc08e084312746ed3ebc14211ba30037d5e44e
Successfully built MarkupSafe
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-
7.1.2 itsdangerous-1.1.0 pika-1.1.0
Removing intermediate container 0873978faef8
 ---> d77de60ef20e
Step 5/6 : ENV FLASK_APP=app.py
 ---> Running in d8af1dd55084
Removing intermediate container d8af1dd55084
 ---> 69ae9b270fb9
Step 6/6 : CMD ["flask", "run", "--host=0.0.0.0"]
 ---> Running in c7c300db2f56
Removing intermediate container c7c300db2f56
 ---> a8f79c7b4cd1
Successfully built a8f79c7b4cd1
```

```
Successfully tagged gcr.io/example-20200428/front-end:v1
PS example-final> docker push gcr.io/example-20200428/front-end:v1 ②
The push refers to repository [gcr.io/example-20200428/front-end]
aae3053b9026: Pushed
07a7dd71e46e: Pushed
62cc2f2db459: Pushed
f3f43710db31: Pushed
e68e29bcc308: Pushed
baf481fca4b7: Layer already exists
3d3e92e98337: Layer already exists
8967306e673e: Layer already exists
9794a3b3ed45: Layer already exists
5f77a51ade6a: Layer already exists
e40d297cf5f8: Layer already exists
v1: digest: sha256:ab9c121705a4f4c47b7e32012d13da96cf0e8e2bc807c49a37b04c2099c7fb2e size:
2636
PS example-final> docker build -t gcr.io/example-20200428/back-end:v1 ./back-end/ ③
Sending build context to Docker daemon   7.68kB
Step 1/5 : FROM python:3.9.0a5-buster
 ---> b5f66cb660dd
Step 2/5 : COPY . /app
 ---> a8faf7d98529
Step 3/5 : WORKDIR /app
 ---> Running in 385122bd0d0e
Removing intermediate container 385122bd0d0e
 ---> 7e517e6b75b1
Step 4/5 : RUN pip install -r requirements.txt
 ---> Running in a28a17d13909
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-cp39-cp39-linux_x86_64.whl
size=498130 sha256=6a715ba7fcf21deca5712a1e404c
51b1b0168ad64c7612890f30935845a6562f
  Stored in directory:
/root/.cache/pip/wheels/c2/17/82/f619fa1d1a361445c4ff28634f734936f2d54891c79840b345
Successfully built psycopg2
Installing collected packages: pika, psycopg2
Successfully installed pika-1.1.0 psycopg2-2.8.5
Removing intermediate container a28a17d13909
 ---> 53827a4ffaea
Step 5/5 : CMD ["python", "app.py"]
 ---> Running in 5aef1da46ef1
Removing intermediate container 5aef1da46ef1
```

```
 ---> 65c9dfe66eb8
Successfully built 65c9dfe66eb8
Successfully tagged gcr.io/example-20200428/back-end:v1
PS C:\Users\rxt1077\it490\example-final> docker push gcr.io/example-20200428/back-end:v1
④
The push refers to repository [gcr.io/example-20200428/back-end]
303c8c71682c: Pushed
50f2e9234064: Pushed
62cc2f2db459: Layer already exists
f3f43710db31: Layer already exists
e68e29bcc308: Layer already exists
baf481fca4b7: Layer already exists
3d3e92e98337: Layer already exists
8967306e673e: Layer already exists
9794a3b3ed45: Layer already exists
5f77a51ade6a: Layer already exists
e40d297cf5f8: Layer already exists
v1: digest: sha256:9446a3d91fa555a84457e51ba2ac3b8e2821f31a531ab21d0e85cd9e37e11dbb size:
2636
```

① Build/tag the front-end image for GCR. Notice how we use the project-id.

② Push the front-end image.

③ Build/tag the back-end image for GCR.

④ Push the back-end image.

We will also need to change the `image` mapping to point to the image on GCR in `back-end-k8s.yml` and `front-end-k8s.yml`. You can see them there, commented out, if you check the source repository.

17.4. Creating Objects

At this point, we should have access to two Kubernetes clusters: our local minikube cluster and a cluster we created in Google Cloud called `example`. Fortunately for us, gcloud has already configured our kubeconfig file for access to the `example` cluster. kubectl refers to these different clusters as `contexts` and we can see what is available with the following command:

```
PS example-final> kubectl config get-contexts
CURRENT NAME                                          CLUSTER
*       gke_example-20200428_us-central1-c_example gke_example-20200428_us-central1-
c_example①
        minikube                                      minikube
```

① We are currently using the Google Cloud context, but if you need to switch contexts, you can use the `kubectl config use-context` command.

We've already pushed our images to GCR and updated the **Front End** and **Back End** objects to reflect

that. Now all we need to do to create objects is use the kubectl commands we're familiar with:

```
PS example-final> kubectl get pod
No resources found in default namespace. ①
PS C:\Users\rxt1077\it490\example-final> kubectl apply -f . ②
deployment.apps/back-end created
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
PS example-final> kubectl get pod ③
NAME                         READY   STATUS    RESTARTS   AGE
back-end-84cd7447d-6j7b7     1/1     Running   0          5s
back-end-84cd7447d-lthqc     1/1     Running   0          5s
back-end-84cd7447d-vlrqz     1/1     Running   0          5s
db-r-5b9977874b-ghj54        1/1     Running   2          13m
db-r-5b9977874b-ngbs5        1/1     Running   2          13m
db-rw-7755dddd76-f9krt       1/1     Running   0          13m
front-end-856bc468cc-dddh4   1/1     Running   0          13m
front-end-856bc468cc-f9ltq   1/1     Running   0          13m
front-end-856bc468cc-rm6dg   1/1     Running   0          13m
messaging-0                  1/1     Running   0          13m
messaging-1                  1/1     Running   0          13m
messaging-2                  1/1     Running   0          13m
PS example-final> kubectl get service ④
NAME         TYPE           CLUSTER-IP      EXTERNAL-IP     PORT(S)              AGE
db-r         ClusterIP      10.47.242.137   <none>          5432/TCP             14m
db-rw        ClusterIP      10.47.252.2     <none>          5432/TCP             14m
front-end    LoadBalancer   10.47.244.185   35.238.67.247   80:30337/TCP         14m ⑤
kubernetes   ClusterIP      10.47.240.1     <none>          443/TCP              21m
messaging    ClusterIP      10.47.253.17    <none>          5672/TCP,15672/TCP   14m
```

① Checking to see what is running initially shows that there are no pods on our cluster.

② Applying all of the YAML files in our current, `example-final`, directory.

③ Running `kubectl get pod` shows all our replicated components.

④ Checking to see what services are running

⑤ The only external, LoadBalancer, service is `front-end`. A quick visit to http://35.238.67.247 will give you front-end access to the system.[5]

We now have a scalable system running on enterprise-grade hardware.

17.5. Cleaning Up

Don't forget that Google Cloud is not a free platform. While we do have free credits to experiment with, the compute and storage resources that we are using are very real. Once we are done, we have to remember to delete our cluster and the resources that our project uses. That can be done with the following commands:

```
PS example-final> gcloud container clusters delete example
The following clusters will be deleted.
 - [example] in [us-central1-c]

Do you want to continue (Y/n)?

Deleting cluster example...done.
Deleted [https://container.googleapis.com/v1/projects/example-20200428/zones/us-central1-
c/clusters/example].
PS example-final> gcloud projects delete example-20200428
Your project will be deleted.

Do you want to continue (Y/n)?  Y

Deleted [https://cloudresourcemanager.googleapis.com/v1/projects/example-20200428].

You can undo this operation for a limited period by running the command below.
    $ gcloud projects undelete example-20200428

See https://cloud.google.com/resource-manager/docs/creating-managing-projects for
information on shutting down projects.
```

17.6. Resources

- Google Kubernetes Engine Quickstart
- Google Cloud SDK Interactive Installer
- Configuring cluster access for kubectl
- Pushing and pulling images

17.7. Questions

1. *Name at least two Google Cloud services that GKE uses.*

2. *What does GCR do and why does GKE use it?*

3. *What is the difference between the gcloud and kubectl commands?*

4. *Compare and contrast the gcloud and minikube commands.*

5. *In the output for* `kubectl get service`*, why is* `front-end` *the only service with an external IP?*

[5] By the time you're reading this, this site should be unavailable. If it's still available, let me know because it's costing me money!