Provisioning and Deployment

Mikael Svahnberg (Mikael.Svahnberg@bth.se)

This is part of a course on Applied Cloud Computing and Big Data, offered by Blekinge Institute of Technology. Version 1.0

Table of Contents

1 Document Overview

The course Applied Cloud Computing and Big Data is a 7.5 ECTS course offered by Blekinge Institute of Technology. The course is organised around three themes, and all three themes must be completed to complete the course:

- Cloud Provisioning and Deployment
- The Business Case for Cloud Computing
- Big Data Analytics

The course is divided across two source code repositories:

- `https://github.com/mickesv/ProvisioningDeployment.git` contains the instructions and source code for the Cloud Provisioning and Deployment, and the Business Case for Cloud Computing parts of the course.
- `https://github.com/mickesv/BigDataAnalytics.git` contains the instructions and source code for the Big Data Analytics part of the course.

This document covers the first two of these themes, i.e. Cloud Provisioning and Deployment, and The Business Case for Cloud Computing. The document is structured as a series of introductions to different topics mixed with hands-on tasks to exercise these topics.

The topic introductions and practical exercises are further punctuated by quizzes that (a) serve as checkpoints of important concepts that have been covered to that point, but also (b) introduce further reading material to deepen your understanding of the cloud computing business case. **These quizzes are not marked**, but serve as a learning aid; for you to summarise what you have learnt so far, and for us to keep track of your progress through the material.

The two themes are examined through a final assignment (See [assignment-build-something], page 64,), where you put what you have learnt so far into practice.

This document is available in different formats:

- as web pages on the course platform
- as a pdf file
- as a TEXInfo tutorial

The TEXInfo tutorial may require a bit more introduction. Access this version e.g. through the command `info Provisioning-Deployment` or in your favourite editor though `C-u C-h i Provisioning-Deployment.info` . Basic navigation in the document is done with the following keys (try `info info` for more details and further key commands):

- <spc> (space) scroll forward
- 'b' Beginning of current node
- 'e' End of current node
- 'n' Next node
- 'p' Previous node
- 'u' Up (usually (but not always) to the top index page)
- <enter> on a cross reference to open it.

2 A Note on Reading Research Articles

This course may suggest research articles for you to read. This may seem a daunting task for you, but there are some general guidelines that may help you:

- Start by reading the abstract and the conclusions. These will tell you what problem or area the article is addressing, and what the article contributed to solving the problem.
- The articles in this course have – to a large extent – been chosen because they summarise the topics in various ways. To find these summaries, look for *lists, figures, and tables* in the article. Read these. Read the text around them, or where the figures and tables were referenced, in order to get an explanation of how to interpret them.

Generally, interpret each section of an article as follows:

- The *introduction* section in an article puts the problem into context, and might give you clues as to how other researchers have solved it before. You can use this section to get a generic overview.
- The *related work* section should discuss in further detail what other researchers have done that is similar to, or relates to, the article you are reading. This may give you an idea of how others have addressed the problem, and may give you pointers to other articles that you would wish to read.
- The *methodology* section explains how the study was conducted. Once an article is published (which they naturally are in this course), this is mostly interesting if you distrust some of the results and want to see if anything has been missed when constructing the study.
- The *execution and results* sections describe how the study was executed (in particular discrepancies from the planned methodology), and what the raw results were. Most of the time, you can safely skip these sections.
- The *analysis* section "bakes" the results and tries to answer the research questions (address the identified problem). This is probably the most interesting section for you to read.
- The *discussion* section should raise the view and try to see what the results actually mean in a bigger context. What can you do with the results? "So what?". This is a tricky section to read, since the authors want you to believe that their results are the best thing invented since hot porridge, but at the same time they have to identify threats to the validity of the study. Validity threats are things that could have influenced the results instead of the sought after effect. This may be things that happened at the same time, inadequacies in the researchers skills and abilities, or inadequacies in the research design. Please remember, when (or if) reading this section, that most of the identified validity threats are minor obstacles (or the article would not have been published) that may impact the scientific view of the results more than the practical significance or usefulness of them.
- The *references* may give you ideas for other articles that you would like to read.

The bottom line is that when being told to read a lot of research articles, the trick is to learn how to not read them while still getting the gist of them. Hopefully, the "map" described above may provide some help in identifying the parts of an article that are important for you.

3 Introduction to Orchestration and Provisioning

A vital part of making the best use of cloud resources is to have a controllable way to set up your cloud environment. Ironically, a large part of this consists of *not* using cloud resources but rather configure a transparent way to set up a local development environment, a potentially local test environment, and the production environment. The goal is to be able to do this with the same command(s) and to make sure that all environments are similarly configured.

You may be used to develop applications in your local development environment. From within this environment you will run and debug your application. You may even have set up automated tests to run as well. There are some challenges with this, however:

- Your development computer needs to remain very stable. If you install some new software or run some update, you can no longer be sure that the application will actually continue to run. Even if it still works on your machine, there is no guarantee that it will also work on the customers' machine.
- When you start working on a new feature branch, your machine is indeed tainted by everything you have developed and tested before. Partially, this is what you have your configuration management system for: You would go back to the main branch and create your new feature branch from there. Technically, this *should* remove everything else you have been doing so that you start from the same page every time. In practice, the settings in your development environment is not affected by what you do with the configuration management tool, your stack of installed software is not affected, and if you use a database it will merrily continue with whatever test data you had in there just before.
- When you are ready to deploy your application (of course you have already tested it?) , you first need to set this server up and make sure that it is installed and setup in exactly the same way as your development machine. And then you deploy. And say a prayer. And hope that no users try to use your application while you are figuring out what went wrong.
- As time goes, you ever so often log in to the server with your running application to install updates, fix minor errors, and upgrade your application. Until things have gone horribly wrong and you need to start from scratch again, and you realise that you have not maintained an updated lab notebook with all the things you have ever done on the server. Things that you have already fixed once re-appear and you have to fix them again. And again.

This brings us back to the overall goal. What you want is a controllable and repeatable way to set up your machines. Controllable, so that you can decide whether you want a development-version (Where perhaps your application is running with debug flags turned on), a test-version (that contains a reasonable subset of all the data in the live system and where tests for different environments are launched automatically) , or a running production version of your system (that you really have no business ever editing directly, and maybe never need to edit directly since it is deployed automatically as soon as all tests are run).

You want the setup to be *repeatable* so that you will always start from the same page, no matter what. Basically, you want to banish even the idea that it is ok to log in to a running environment to change things. If you want to introduce changes, you change the repeatable scripts, test them, and then push them to the rest of the project for common use. Now, *anyone* in the project can create the exact same environment as you have, and test your code under exactly the same conditions.

Today, you would not even dream of working in a larger development project without having some form of configuration management support. You create feature development branches, merge features into testing branches, which eventually are merged into a main branch. Your automated tests are treated in the same way; The test code may even be part of the same commits as the application code. By creating scripts for your deployments, you now have the ability to

put these too under configuration management. No more guesswork about which version of framework "frobnicator" that was launched together with version 1.3.42 of your application – it's in the setup scripts.

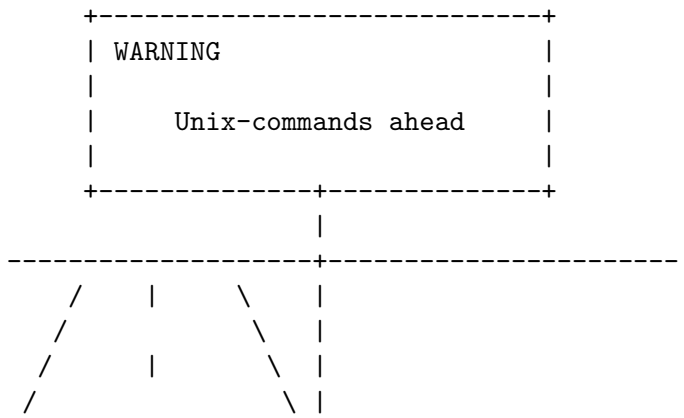
Summary

The *ultimate* goal remains to be able to deploy your application to a cloud server somewhere. But the *path to get there* require you to spend time and effort to set up a repeatable foundation to develop locally. As always, by laying a proper foundation it is much easier to reach the end goal.

In this part of the course we are thus going to be looking at different ways of setting up local development in a controllable and repeatable way. We are going to look into different types of deployments (e.g. microservices or infrastructure-as-a-service), and different types of deployment targets (local, test, production).

We are going to do this through a combination of theoretical knowledge and practical exercises. The practical exercises will allow you to try out different tools for creating different types of deployments.

A word of warning There are many tools to master, and each of them use their own language. We are going to be working in a console (Command Prompt, terminal, xterm, or whatever your operating system provides) using the command line interfaces of these tools. The reason for this is that we want to create a solution that can be combined into a bigger scripted solution, and this is simply not possible if you are expected to intervene and point-and-click your way to a solution.



3.1 Learning Outcomes

Relevant learning outcomes from the course syllabus are:

Knowledge and understanding On completion of the course, the student will be able to

- In depth be able to describe different types of cloud platforms
- In depth be able to describe different reasons for adopting a cloud solution, and the challenges with these different reasons.
- In depth be able to reason about solutions to the common challenges with the cloud solutions.

Competence and skills On completion of the course, the student will be able to:

- Independently be able to set up a development environment consisting of local machine configurations and cloud based servers.

Judgement and approach On completion of the course, the student will be able to:

- Be able to evaluate different reasons for choosing a cloud solution and select a suitable solution accordingly accordingly.

4 Microservices and Lightweight Containers

As applications grow, it becomes more and more difficult to understand all of them at once. The programming solution is to divide the application into components, where each component has well defined responsibilities and interfaces. Component-based software engineering and code modularisation (and – as it is most often used – object oriented software development) is an answer to the need for smaller, more manageable parts of a larger system. However, ultimately, the main reason for this type of modularisation is *development* needs.

Going further down the rabbit hole of software architectures, we do see that there are other reasons for modularisation. One often discuss different viewpoints of a software architecture (which – just like UML – has its origins in P. Kruchten "The 4+ 1 view model of architecture." *IEEE software* 12.6 (1995): 42-50.), with a *conceptual view* to understand what to build, a *module view* to understand how to structure it while building, and an *execution view* to understand the threads, processes, processors, memory, and shared memory required in order to run the system. These are not created in isolation; the module view in particular need to reflect both the conceptual view as well as the execution view in order to build the right execution targets so that the application can be properly deployed.

Thus: as an application grows in size and complexity, there is a need to structure it into modules for the sake of conceptual understanding and for the sake of scalability.

Some types of execution modularisation is probably already familiar to you. You have probably heard of *client-server architectures*, and you have probably built systems using an external database. If you are unlucky, you may have developed multi-threaded applications.

Microservice architectures is an attempt to simplify development. On an architecture level, it is an attempt to settle on a single type of interface between the different conceptual components (usually a Section 4.7 [REST API], page 11), align the execution of an application with its conceptual view, and defer creation of the module view to each individual component/executable. Of course, this works less well in practice because one still wish to re-use components between the different microservices but that's another story.

One more benefit (some would argue it is the main benefit) is that there is no longer any need for multi-threaded programming. Each microservice can (and perhaps even should be) as single-threaded as possible. If one needs to scale the application this is done as part of the deployment rather than with the help of logic inside the application. This creates a *separation of concern* between application logic and deployment logic which is highly sought after.

Another benefit is that each microservice can be written in the way and language best suited for its particular responsibility. Some may use a particular framework that dictate how to structure the code, others may be written in some high performance language, and others again may consist of a simple stringing together of a couple of Unix commands.

A third benefit is that with the separation between application logic and deployment logic it is easy to attach monitors to each microservice to gauge e.g. whether it is running, healthy, or overloaded. Logs can be created and inspected within each microservice, and temporary or permanent storage may be attached to each microservice.

4.1 Building Microservices

What do we then need to do in order to build our application as a collection of collaborating microservices?

In the most basic level, we simply build our application as a collection of executables. We launch each executable as a separate process, and have some mechanism to communicate between the processes (e.g. sockets, remote procedure calls, pipes, file system, or through the database). Challenges are that we still have to start and connect all the different parts of our system when

running, and we need to build the logic ourselves to scale up or down. If we want to scale one part of our application by starting more processes of the same type, we need to manage the logic for this. If we want to "scale out" to another machine, we need to implement and manage the logic for this. If one process writes to a particular file, all other processes are directly influenced by this, and need to have logic implemented to avoid conflicts.

Next level up, at least on UNIX systems, is through '`chroot`'. Through this program we basically "sandbox" the file system for a particular process so that we can have separate parts of our disk allocated to different processes (called a "chroot jail"). Within this new root environment we can install tools only needed for this particular process. We can also have separate sections of our disk for e.g. testing environments and development environments, and we can isolate all dependencies from one process to only that which is available within its particular chroot jail.

'`chroot`' is old, it was created in 1979, and as computers became more powerful it started to be used as a lightweight virtualisation. There are, however, some considerable limitations. For the sake of this course, the most major limitation is that any logic surrounding the use of chroot – scaling, inter process communication, etc. – still has to be implemented by the developers of the application. Shirley there's an app to help with that!

As it happens, there are applications for this. You may have heard about Docker `https://www.docker.com/` or its open source cousin Podman `https://podman.io/` . Both of these are examples of Linux Containers (LXC), which are usually described as "Lightweight virtual machines". With tools like these you create a *Container* with a specific set of programs installed, and then you can manage this container as a whole, rather than having to keep track of each individual part inside.

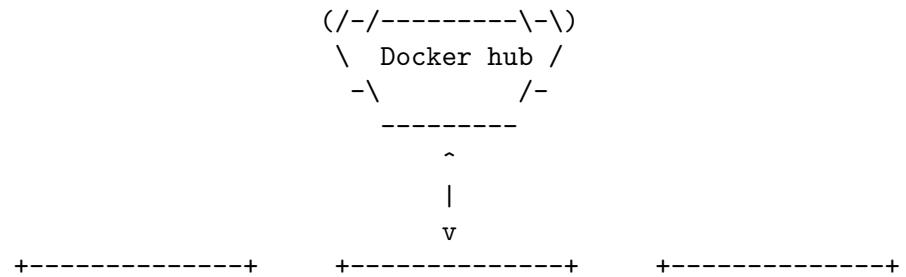
This container essentially consist of two things: (a) The software needed to run your microservice, and (b) your microservice, and the idea is to keep both of these as small as possible. Do not expect a full user experience from the software stack in a container, be prepared to fight with `sh` rather than something slightly easier to use like `bash`. As for the microservice that you deploy in the container, try to maintain a strict responsibility-driven design: If you want to give a container two responsibilities then consider splitting it into two containers.

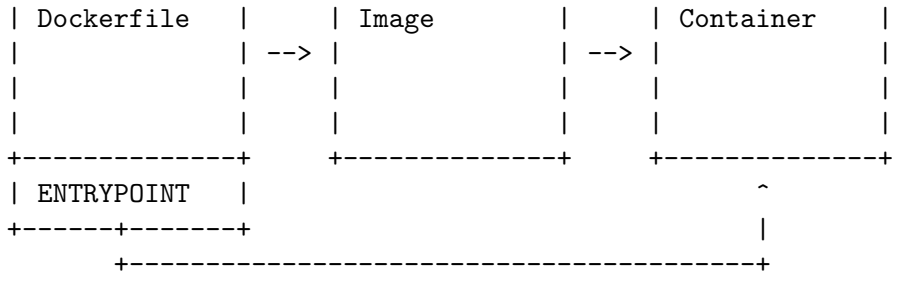
4.2 A Quick Rundown of Docker Terminology

When working with Docker you start with a `Dockerfile` . Each line in this Dockerfile creates a `Layer`, and Docker will try to be smart about caching these layers so that if you re-use your Dockerfile it will start from the last layer that is the same as one it has previously built.

Based on this Dockerfile you create an `Image` . This is a snapshot in time from when all the lines in the Dockerfile have been executed. You can version control and tag this image if you like, and it is this image that you will push to Docker hub `https://hub.docker.com/` . If you are using a ready-made container from Docker hub, e.g. with a database or a web server installed , it is this image that will be downloaded and used as a layer in your own Docker image.

When you start your Container, it will use the image as a base and execute the `ENTRYPOINT` in the Dockerfile, which is the command and its parameters that should be used to start your microservice. You now have your running Container.





4.3 Dockerfile

Typically, a Dockerfile will contain at least the following commands (Most of which can appear more than once):

`FROM` to specify which image to use as a base for the container. This can either be an image you have created yourself, or a ready-made image from Docker hub. Example: `FROM node:18-alpine` says that you want to use an image that has node.js version 18 installed on top of an Alpine Linux install (which is a lightweight Linux distribution, more than which you probably do not need).

`RUN` , which has a whole range of different parameters (`https://docs.docker.com/engine/reference/builder/#run`), but you would typically use this to add any programs that you need but which were not part of the original image. Example `RUN npm install -g nodemon` means that you wish to use npm (one of the package managers available for node.js) to install the program `nodemon`. Nodemon is used during development to restart a node.js application as soon as any of its files change.

`EXPOSE` is used to flag network ports that can be of use outside of the container. Example: `EXPOSE 3000` flags port 3000 as being accessible.

Please note: This only instructs Docker that this port *can* be opened. You still have to explicitly tell Docker that you *want* it open and mapped to a particular port on your host machine when starting your container.

`WORKDIR` instructs Docker that when you start the container, this is the directory it should start in. Example `WORKDIR /app` to start in the `/app` directory.

`COPY` is used to move any files from your host machine into the image. You would for example use this to copy the code for your microservice into the container. You can also use it to copy a specific configuration etc. Example: `COPY . .` copies everything in the current directory on the host machine (probably the directory where your Dockerfile is located) into the current working directory inside the container (which you may have set with a previous `WORKDIR` command).

At this point you are in many cases done with what needs to be installed, what remains is perhaps to set a few environment variables, and instruct Docker how you want to start your microservice. If you are creating a node.js application, this is a good point to ensure that all node.js packages you depend on are also installed. Another RUN takes care of this: `RUN npm install`

Setting an environment variable so that node.js, when running, knows what debug output to focus on: `ENV DEBUG='qfapp:*'`.

`ENTRYPOINT` is typically the last line in your Dockerfile, and it specifies which command to run and which parameters to use for it. You can specify a `CMD` instead (or together with `ENTRYPOINT`), but it is generally recommended to try with only an `ENTRYPOINT` first, since this can be replaced even after the image is created (e.g. if you realise that you want to try out a different start command). The preferred format (which can be used for other command such as `RUN` too) is to use an array format for the command: `ENTRYPOINT ["npm", "run", "dev"]` .

Other commands that you sometimes see are used to mount VOLUMEs, or set the USER. Please see the full reference for the Dockerfile `https://docs.docker.com/engine/reference/builder/` for more information.

```
FROM node:18-alpine
RUN npm install -g nodemon
EXPOSE 3000
WORKDIR /app
COPY . .
RUN npm install
ENV DEBUG='qfapp:*'
ENTRYPOINT ["npm", "run", "dev"]
```

4.4 Docker Commands

Now that you have created your Dockerfile, the next step is to start using it.

1. Getting Help

   For almost all commands in Unix, you have a few different ways to get help. Running the command with the parameter '`-h`' , or '`--help`' (note the two dashes) is a good start. For more detailed information, you use the command '`man`' . Sometimes, you can get even more details if you use the command '`info`'. Try the following commands:

   ```
   docker -h
   docker --help
   ```

   Wow, that was a long page. Let's pipe it through something that paginates it for us. We can use either '`more`' or its more enabled version '`less`' for this.

   ```
   docker -h | more
   docker --help | less
   ```

   To see how much more information you can get, try the following commands (you can also learn more about '`man`' by looking at *its* man-page: '`man man`'):

   ```
   man docker
   info docker
   ```

   Pressing the key '`h`' while on a man-page gives you instructions on how to navigate the document.

2. Docker image and Image Management

   First, we need to use the Dockerfile and create an image. This image downloads everything that should be installed – provisioned – in order to run your application later. During this process, all commands in your Dockerfile will be executed, and the image will be stored ready for use. Note that once this step is complete, nothing is actually running. All you have is a chunk of disk space that is *ready* to run.

   The command is '`docker build`' , and we want to look in the current directory for the Dockerfile so we add a period to the command. However, if we do this we get an image which has no name and only an unwieldy image id to work with. So let's tag the image as we create it with the '`-t <tagname>`' flag.

   ```
   docker build -t myfirstimage .
   ```

   Did it work? Let's view all images:

   ```
   docker image ls
   ```

   The output should be something like:

   ```
   REPOSITORY                  TAG         IMAGE ID       CREATED          SIZE
   myfirstimage                latest      270970b0971b   5 seconds ago    232MB
   ```

What else can we do? Try '`docker image --help`' . Apparently we can (among other things) build, list, inspect, remove, and tag images with this command. '`docker build`' is, in fact, an alias for '`docker image build`'.

Once you are done (not yet), you may want to clean up after yourself. Each image has a name listed under REPOSITORY, a tag listed under TAG, and an IMAGE ID. If there is only one image with that name, you can run '`docker image rm myfirstimage`' . If you have several images with the same name (try creating one more with '`docker build -t myfirstimage:second .`') , you will need to qualify the name with the specific TAG you want to remove. If all else fail (e.g. if you forgot to give the image a name in the first place), you can use the IMAGE ID to uniquely identify your image.

3. Docker run and Container Management

   So, let's get this image to actually run as a container, using the command '`docker run`'. We need a few parameters for this to work as we want it. Some of the more commonly used parameters are listed in Table Table 4.1

Short Parameter	**Long Parameter**	**Description**
-d	–detach	Run the container in the background without any console interaction
-p	–publish	List the ports that should be exposed from inside the container to a speci
-e	–env	Set environment variables inside the container
-v	–volume	Mount a volume (or directory) from the host is into the container
	–name	Give the running container a specific name
-i	–interactive	Run the container as an interactive console application. Usually together
-t	–tty	Allocate a terminal (console). Usually together with -i

Table 4.1: Some of the more common parameters to docker run.

The most common ones you will see are the detach and publish flags, and you will often see these together: '`docker run -dp 8080:3000`' , which will open port 8080 on your host into port 3000 inside the container.

When developing your application, it is useful to be able to view any console output it makes (e.g. if you use '`printf()`', '`cout`', '`console.log()`' , or '`(print)`' inside your application). '`docker run -it`' enables this, and also enables you to give commands into your running application if it is reading from standard input.

You can view running containers with '`docker container ps`', '`docker container ls`', or just '`docker ps`' :

```
CONTAINER ID   IMAGE     COMMAND          CREATED          STATUS          PORTS      NAMES
988c2fa79431   tst       ``npm run dev''   7 seconds ago   Up 6 seconds   3000/tcp   per
```

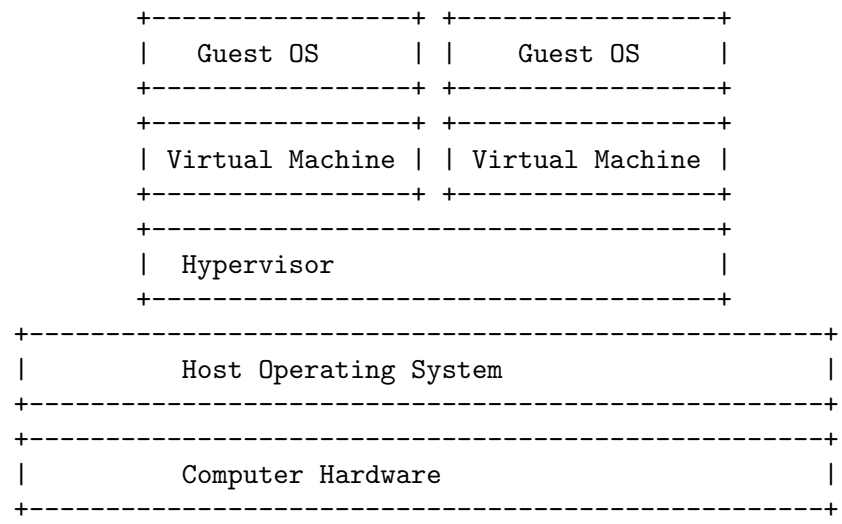
the CONTAINER ID is used as a handle to do other things with the container. We can, for example, stop the container and later start it again with '`docker container stop <container-id>`' and '`docker container start <container-id>`' . If we stop the container we can also remove it completely (e.g. in order to update with a newer image): '`docker container rm <container-id>`' (adding the "force"-flag '`-f`' to rm will first stop the container if it is running, saving you one command: '`docker rm -f <container-id>`').

4.5 Not Building Microservices – Infrastructure As A Service

The alternative to running microservices would be to run a full-fledged virtual machine with its full operating system and everything. Maybe even a graphical user interface. Sometimes this is the better option, especially if an application consists of several collaborating processes that expect to co-exist on the same machine. We *could* run these inside a docker container too, but

this would sort of break the minimalistic view that one container should essentially only run a single command.

In order to run virtual machines, you need a Hypervisor software that can act as a management layer between your host operating system and the guest operating system that you want to run in your virtual machine (Technically, I suppose that Docker/Podman should also be considered hypervisors). The virtual machine is supposed to emulate the hardware, so you are free to – in theory – run any operating system built for any platform. In practice, however, there are limitations to this.



Some common hypervisors include:

- Virtualbox `https://www.virtualbox.org/`
- VMWare, with VMWare Fusion for Mac `https://www.vmware.com/products/fusion.html` , and VMWare Workstation for Windows and Linux `https://www.vmware.com/products/workstation-player.html`
- Parallells `https://www.parallels.com/se/` if you are using a Mac
- Hyper-V `https://learn.microsoft.com/en-us/virtualization/hyper-v-on-windows/` if you are using Windows

Of these, Virtualbox is free for use and available on most operating systems and mostly works without problems. It is, however, limited to x86 and AMD64/Intel64 hardware architectures so if you are using e.g. a more recent Mac you will not be able to use it. In those cases, you *can* make do with VMWare or Parallells, but each of them have their own unique set of error messages and challenges to overcome in order to work seamlessly.

Qemu `https://www.qemu.org/` is a more powerful and free alternative that actually does emulate the virtual hardware as well. So you can emulate an x86 architecture on top of a mac M1 machine, or vice versa. The installation and setup *looks* more daunting than e.g. Virtualbox does, but it is really quite easy – at least on a Linux machine. In my experience, Qemu is also much faster than a corresponding virtual machine in Virtualbox.

4.6 Provisioning your Virtual Machine

With a hypervisor such as above, you get a clean virtual machine, and it is up to you to install whatever operating system you want to run, and whatever software you want installed on top of this operating system. You *can* do this manually, but if you remember the goals we set up in the Introduction (See [introduction], page 3,) , we want to do better than this. We wish help from the computer to create a repeatable and configuration managed deployment that can be used over and over again.

This is where the software Vagrant `https://www.vagrantup.com/` comes into play. With Vagrant we create a file '`Vagrantfile`' where we specify the size and type of virtual machines we want, the operating system we want on them, and any other software we want installed – analogous to the '`Dockerfile`' discussed earlier, but with much more details. It is important to note that Vagrant does not create any virtual machines itself, it is not a hypervisor. Instead, it interacts with other hypervisors such as Virtualbox, VMWare, Hyper-V, or Qemu and requests them to do the "heavy lifting" for it. The beauty of this is that you learn once how to work with Vagrant, and can use the same Vagrantfile to create your same virtual machine using any of the supported hypervisors and indeed even launch your virtual machine on a cloud provider.

Unlike Docker, a Vagrantfile can also specify several machine configurations in one, so you can Orchestrate your entire application at once instead of having to work separately with each component/microservice in your system setup. But this is another story which we will get to later (see [Deployment], page 45,).

4.7 Communicating Microservices and REST APIs

With microservices we are tearing apart our monolith application into several smaller units. Where we previously could use a simple method call inside the same process, we must now reach for other tools. Since these smaller units need not run on the same (virtual) machine, we must also resort to communication means that are network-enabled. If we truly love pain, we may develop our own communication protocol and use simple sockets, and for some applications this may be the best way to establish a fast enough and secure enough communications means. One step up, we may rely on Remote Procedure Calls (which still use sockets but we do not need to see them).

What is more common today, however, is that we make use of ready-made components to turn our microservice in to a small web server. The HTTP protocol is already well established and provide support for most of the tasks we need (i.e., GET, POST, PUT, PATCH, and DELETE). The payload of the communication protocol still need to be defined, of course, but much effort can be saved by sticking to existing grammars such as JSON or XML.

This leads us to Representational State Transfer, or REST APIs for short. Key principles for a REST API are (source: `https://www.ibm.com/topics/rest-apis`):

Uniform Interface
: All requests for the same resource should look the same. So we cannot embed state or cookies or user session or whatever in the address of the resource (but we can include such information in the actual request)

Client-Server Decoupling
: Clients and Servers are independent from each other, all we need to know is the address of a requested resource. In fact, this is all that is ever available to us.

Statelessness
: This one is interesting given the original description of the REST pattern as transitioning a service through a series of states. The key here is that the server does not keep track of the state for every client that is connected, this is instead made part of each client's requests.

Cacheability
: Whenever possible, the same request should give the same answer (this may not be desired if, for example, content is being continuously generated, and in those cases the resource should flag itself as being not cacheable).

As already almost stated, over time we have mostly settled on using HTTP (or HTTPS) as the communications protocol and JSON or XML as the data protocol for our REST APIs.

Because there are ready-made components for many programming languages today, it is easy to start listening for HTTP calls and to parse JSON data. Creating HTTP calls is equally easy, and so we have an easy mechanism for providing and using a service that works across system borders.

The downside is that we break the seamlessness of our application. Ideally (and this was once the sales pitch for remote procedure calls), we want to make all function/method calls in the same way throughout the entire application, no matter the physical distribution of parts of the application to different servers. In fact, the first thing we do when we create a microservice is probably to define a *Layered* architecture style where the top layer deals with all the HTTP/JSON complexities, so that the rest of the application can be an as pure representation as possible of the domain concepts. Likewise, when we develop clients that need to make REST calls to another service, we will probably create a layer that marshal regular function calls into JSON/HTTP requests and unmarshall the response into a return value that knows how to behave like a proper object.

4.8 Other communication means

Do note that the required statelessness of a REST service means that the communication should only ever be one-directional. A well behaved service dutifully waits until someone makes an HTTP call, responds, and then goes back to waiting for the next call. If we expect the service to call back at a later stage (perhaps when a long-running job is done, or when a monitored resource becomes available, or when a specific user comes online, etc.), there should not be any way for the service to even know about the clients. And yet sometimes we need this, which means we sometimes need to break the simpleness of our REST service.

First, we may need some mental acrobatics to decide that a request is not done until we have called back to the originator with results. If the initial request contains the return address, we can pretend that we are still working from within the same request and we are still (please click your heels three times and believe very firmly) working within the conceptual model of the REST architecture pattern.

One challenge is that we may not be able to spawn new threads inside our application, e.g. if the programming language or paradigm does not support threads. Thus, if the client makes a synchronous call, our hands are tied and we must deal with the execution thread we have been given to us, process the request, and return the answer. Ideally, we want something like:

1. A call is made to our REST service together with a callData object.
2. We store the callData object.
3. We prepare a response to the REST call and write this to the open network socket.
4. We close the network socket but DO NOT return from our method just yet.
5. Now, we can start processing the callData object and figure out what they really wanted us to do.
6. Once we have prepared our more detailed response, we look in the callData object and locate the address we should use for our return call.
7. *We* make a REST call to the client and drop off our carefully prepared response.
8. And now, finally, we can return from our method inside the REST service.

. . . Assuming, of course, that the client is accessible through layers and layers of firewalls and NAT:ed networks. I think you get the idea that this is not precisely a neat and tidy way of programming. Nor is it easily supported by common web frameworks, which usually expect steps 3 and 4 to be done by returning from our method, and thus giving back the execution thread.

Our next option is to coerce the client to make an asynchronous call to us. This is tricky using the standard HTTP api. A message queue is a better alternative, and is something which

we can (and will have to) do by introducing a separate microservice in charge of the message queues. The workflow is now:

1. The client drops off a message in the message queue and returns.
2. The REST service gets notified that there is a message to process, or simply polls at regular time intervals.
3. The REST service collects the next message to process.
4. The REST service process the message and prepares a response.
5. The REST service looks up the return address in the message, and returns the response accordingly.

We're getting there. Now all we need to do is decide on how the REST service should contact the client. As before, the client can provide a REST api of their own, in order to receive the return call. Or, since we now have a microservice for managing message queues in place, we can use this:

1. The client opens up a return queue
2. The client drops off a message (containing, among other things, the address to the return queue) in the outgoing message queue and returns.
3. The client checks the return queue until a message arrives.

Having established that this is indeed possible, the question remains: Is this a good idea? Please discuss this in small groups.

The message queue may be implemented in many ways. We may, for example, use a dedicated table in our database (to which our entire system is connected anyway). We may dedicate a separate microservice running e.g. Redis `https://redis.io/` , and use an implementation of a message queue that connects to this microservice. We may even wish to go formal and pick a service that implements the MQTT protocol (Message Queue Telemetry Transport protocol). But that's a story for another day.

5 Quiz: Hypervisors and Lightweight Virtual Machines

TODO Please read the following articles:

1. Crosby, Simon, and David Brown. "The Virtualization Reality: Are hypervisors the new foundation for system software?." Queue 4.10 (2006): 34-41.
2. Morabito, Roberto, Jimmy Kjällman, and Miika Komu. "Hypervisors vs. lightweight virtualization: a performance comparison." 2015 IEEE International Conference on cloud engineering. IEEE, 2015.
3. Jiang, Congfeng, et al. "Energy efficiency comparison of hypervisors." Sustainable Computing: Informatics and Systems 22 (2019): 311-321.
4. Riddle, Andrew R., and Soon M. Chung. "A survey on the security of hypervisors in cloud computing." 2015 IEEE 35th International Conference on Distributed Computing Systems Workshops. IEEE, 2015.

We have chosen these articles for you to read because they illustrate first that there are different types and levels of hypervisors, and secondly because they illustrate that there are many different concerns to consider when selecting which hypervisor (and hence, indirectly, which cloud provider) you chose. The articles above measure and discuss performance, energy consumption, and security.

TODO Summarise each article (no more than 1/2 page each) according to the following:

- Authors and Title of the article
- Briefly, what is the article about?
- What have they measured?
- What are their main findings?
- What can you learn from this article?

TODO Answer the following questions:

- What is a hypervisor?
- What are the main differences between a lightweight virtual machine and a virtual machine?
- What is a microservice?
- What is the REST architecture pattern? How is it implemented in modern microservice development?
- What is "Infrastructure-As-A-Service" (IAAS)?
- What are the main differences between microservice development and IAAS?

There is a Quiz on Canvas where you can submit your article summaries and answers to the questions.

NOTICE: *This quiz does not contribute to the grade in the course. We do, however, require of you to submit the quiz on time.* The purpose of this quiz is to serve as a learning aid allowing you to think about these concepts, and for us to keep track of your progress in the course. If you are unable to maintain the study pace required to submit this quiz on time, we want to be made aware of this so that you are able to re-plan your commitment for the remainder of the course.

6 Let's get Practical

6.1 Introducing the QuoteFinder application

QuoteFinder is the application we are going to use for testing purposes. This application consist of a web site where you can search for quotes in texts. *Version 1*, which we are starting with, simply searches for the entire string in its entirety. In *Version 2* and *Version 3*, this basic search algorithm is replaced by one that allows for a more advanced search where the words may appear near each other if not directly together. We will initially stick to version 1, as it is mostly a *standalone* version. We will later come back to version 1 when we explore other concepts. In this part of the tutorial we are going to, step by step, install and run QuoteFinder version 1 in different ways, but first let us explore the application a bit.

QuoteFinder Version 1 has the following characteristica:

- it is written in node.js `https://nodejs.org/`
- it is an Express web app `http://expressjs.com/`
- it also uses socket.io `https://socket.io/` to communicate between the web client and the express server
- it uses a MongoDB database `https://www.mongodb.com/`

Once started, there are three landing pages:

- `/` This is the start page, where you enter text to search for.
- `/add` From the `/add` page you can add new texts to the database. For adding texts, we recommend Project Gutenberg `https://www.gutenberg.org/` . Indeed, if you leave the text fields empty, you will add a plaintext copy of Leo Tolstoy's *War and Peace* to your database.
- `/list` Last but not least, `/list` allows you to view a list of currently added texts.

Being an Express app, this is all controlled by so called routes, which are set up in the file `src/index.js`:

```
var router = express.Router();
router.get('/', startPage);
router.get('/add', addTextPage);
router.get('/list', listTextsPage);
app.use('/', router);
```

The second argument for each of the `router.get()` calls is a reference to a function, also found in `index.js` . These are fairly small, and mostly rely on jade/pug to render a page back to the user. The `listTextsPage()` is slightly longer since it first creates a new `TextManager` object (more on this class later) to find all the available texts from the database.

```
function startPage(req, res) {
    console.log('Loading start page from: ' + req.hostname);
    return res.render('index', serverIDMessage);
}

function addTextPage(req, res) {
    console.log('Serving the "Add New Text" page...');
    return res.render('addText');
}

function listTextsPage(req, res) {
    console.log('Listing available texts.');
```

```
        let tm = new TextManager();
        return tm.connect()
            .then( tm.listTexts )
            .then(texts => res.render('index', {textList: texts,
                                                ...serverIDMessage}) );
    }
```

Jade/Pug is out of scope for this brief walk-through, but the templates used to render the pages can be found in the directory `/src/views` . One more thing to point out here is the use of *Promises* rather than the usual callback hell that you often end up in with JavaScript (the `then()` - daisy-chain is a clue that Promises are being used).

The rendered web page connects back to the express server using *socket.io* , so when a button is pressed in the web browser, this does not generate a new HTTP call back, but rather just a message on an already open communications socket. This seamlessly extends the event-driven architecture of node.js (with `EventEmitter.emit()` and `EventEmitter.on()`) to also work across system boundaries to the web browser. In the directory `/src/public/js` you will find the two client-side javascript files that takes care of this.

This design decision was taken so that the web page could be rendered in pieces. For example, when a search is being executed, partial results can be added to the page while still waiting for more complex calculations. It will, however, cause problems for us down the line, so please remember that you have been forewarned.

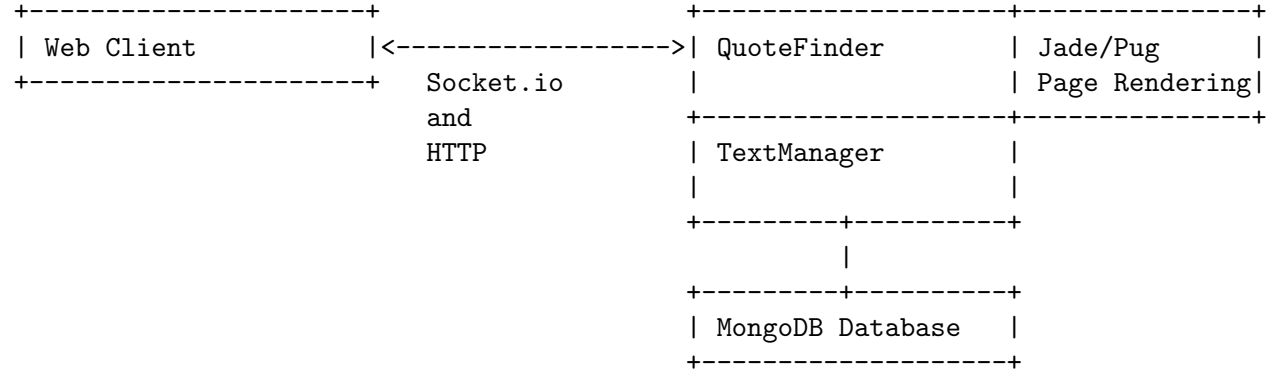
Because the socket object is required in order to send results back to the web client, the methods that require this (i.e. `searchTexts()` and `addText()`) are declared once for each connection. That is also when the server-side socket is connected to pass on messages from the client to these two methods:

```
socket.on('search', searchTexts );
socket.on('addText', addText)
socket.on('disconnect', () => { console.log('user disconnected'); });
```

. . . And that pretty much takes care of `index.js`, with some excursions into views and client side code. The remainder is mostly Express boilerplate code to deal with simple error handling and to actually start listening for connections.

The other component needed is the TextManager class, which is responsible for connecting to MongoDB, and searching, adding, and listing texts. The connection can be done in two different ways, but for now we will only be using the second alternative where (optionally) an environment variable `TEXTSTORE_HOST` decides the IP-address for the MongoDB server. If left unspecified, TextManager assumes localhost (127.0.0.1).

A conceptual view of the application:



6.2 Install relevant software

Before deploying the QuoteFinder app, we need to install and familiarise ourselves with the technology stack we are going to use. Specifically, this includes the different types of hypervisors that we are going to use, and supporting software for these:

- (Windows and OSX) Docker Desktop
- (Linux) docker
- (Linux) docker-compose
- (Linux) minikube and kubectl
- Virtualbox or Qemu
- Vagrant

TODO Install Docker `https://docs.docker.com/get-docker/`

On Windows or OSX, the software to install is *Docker Desktop*. This is now available on Linux as well, but I suggest that you nevertheless install the command line version `docker` via your regular package management tool. (On a mac you can use `homebrew` for this `https://formulae.brew.sh/cask/docker` , which enables you to reproducibly script the installation of your software).

On Linux, you will want to install `docker`, `docker-compose`, and `minikube`.

TODO Install Vagrant `https://developer.hashicorp.com/vagrant/downloads`

TODO Install a Hypervisor Vagrant provides a script language for provisioning and orchestrating a solution, but it relies on that you already have a hypervisor installed (see Section 4.5 [Not Building Microservices – Infrastructure As A Service], page 9,). Virtualbox `https://www.virtualbox.org/` is free and easy to install if your hardware is supported. Vagrant works more or less straight off with Virtualbox.

You may wish to try Qemu `https://www.qemu.org/download/#linux` , and all its gory details `https://wiki.archlinux.org/title/QEMU`. If you do, you will need the following two vagrant plugins, and config the libvirt provider in the '`Vagrantfile`' (see):

```
vagrant plugin install vagrant-mutate # to convert boxes between different hypervisors
vagrant plugin install vagrant-libvirt

# You may also wish to download and convert a box image for use with qemu:
vagrant box add bento/ubuntu-20.04
vagrant mutate bento/ubuntu-20.04 libvirt
```

TODO Run some Tutorials To familiarise yourself with Docker and Vagrant, please work through some of the tutorials they provide:

1. Docker `https://docs.docker.com/get-started/`
2. Docker with node.js `https://docs.docker.com/language/nodejs/`
3. Vagrant `https://developer.hashicorp.com/vagrant/tutorials`

6.3 Installing and Running QFStandalone

1. Full Virtual Machine

   The first step is to run the QFStandalone app as if you were running it on "bare metal", i.e. just as you would do if you install it directly on your machine. The benefit of running it in a separate virtual machine is that you will not litter your own machine with any of the software required for this one app. If you are involved in several different projects, each requiring their own technology stack and (possibly) different versions of some tools, it quickly becomes a nightmare to keep track of on a single machine. If you instead get

into the habit of always developing inside a virtual machine, you will avoid much of this pain. Not to mention that you should not trust the software that some random teacher tells you to install, so running things inside a virtual machine is a prudent security measure. Lastly, should you later decide do deploy to a cloud provider, your machine setup is already scripted, configuration managed, and ready to go.

Tasks

0. Open a terminal in whichever way your operating system expects you to.
1. Create an empty directory on your machine, e.g. `QuoteFinder/Version1/vagrant` , and cd into it.
2. Create a Vagrantfile according to what is required for your particular hypervisor.
   - The box to use is '`bento/ubuntu-20.04`'
   - Define one VM, e.g. called '`app`'
   - Configure this VM so that it forwards the guest port 3000 to the host port 8080
3. Set up provisioning of the VM to install MongoDB and node.js as follows:

```
config.vm.provision "shell", inline: <<-SHELL
sudo apt-get update
sudo apt-get install -y git curl wget gnupg

echo "----------> Installing MongoDB-org"
wget -qO - https://www.mongodb.org/static/pgp/server-6.0.asc | sudo apt-key add -
echo "deb [ arch=amd64,arm64 ] https://repo.mongodb.org/apt/ubuntu focal/mongodb-or
sudo apt-get update
sudo apt-get install -y mongodb-org
sudo systemctl start mongod

echo "----------> Installing node.js"
curl -fsSL https://deb.nodesource.com/setup_19.x | sudo -E bash - &&\
sudo apt-get install -y nodejs
npm install -g nodemon
SHELL
```

4. Set up provisioning of the VM to download QuoteFinder and set it up as follows (note that this is run as the user '`vagrant`' in order to get file permissions right):

```
config.vm.provision "shell", privileged: false, inline: <<-SHELL
echo "----------> Downloading QuoteFinder"
cd /home/vagrant
if [ -d "./QuoteFinder" ]; then
   echo "repo already cloned..."
else
   git clone https://github.com/mickesv/QuoteFinder/
fi
cp -r QuoteFinder/Containers/Version1 .

echo "----------> Setting up QFStandalone"
cd Version1/QFStandalone
npm install
SHELL
```

The if-statement here is not strictly necessary but it helps us if we want to re-provision without destroying the VM first, since otherwise git will throw an error at us.

5. Start the machine: `vagrant up`

6. Log in to the machine with `vagrant ssh` and double check that everything is installed as it should be:

```
$ node --version
v19.7.0
$ mongosh --version
1.7.1
$ ls -l Version1/QFStandalone/
total 116
-rw-rw-r--   1 vagrant vagrant    150 Feb 23 11:37 Dockerfile
drwxrwxr-x 134 vagrant vagrant   4096 Feb 23 11:38 node_modules
-rw-rw-r--   1 vagrant vagrant    414 Feb 23 11:37 package.json
-rw-rw-r--   1 vagrant vagrant 100895 Feb 23 11:38 package-lock.json
drwxrwxr-x   4 vagrant vagrant   4096 Feb 23 11:37 src
```

Your specific version numbers may of course vary, but specific things to look at is that the node version is not something hopelessly outdated such as '`0.10.0`', that the owner of the files is '`vagrant`', that there is a '`package-lock.json`' file, and a '`node_modules`' directory.

7. log out and start the QuoteFinder app with `vagrant ssh -c 'cd ~/Version1/QFStandalone && npm run dev'`
8. We can now test the app. Open a web browser to `http://localhost:8080/add` and click the 'Add' button to add a book to your database. Your terminal window where the app is running will print some info, hopefully ending with *Text added.* If not, make note of any error messages printed.
9. Go to `http://localhost:8080/` and search for something, e.g. 'prince'. This will give a list of hits. Remember to keep an eye on the output in the terminal (ignore the '`TEXTSTORE_HOST`' warning for now).
10. We can also test editing the app. In a new terminal, on your *host* machine, navigate to the directory where you have your Vagrantfile and amaze at the fact that this directory still contains nothing but the Vagrantfile.
11. Log in to the running virtual machine with `vagrant ssh`, change directory to '`~/Version1/QFStandalone/src`' and open the file '`index.js`' in an editor. At least two editors are installed by default, '`vi`' and '`nano`'.
12. In the function `startPage()` , instead of directly return a page rendering, let us by default always list the available texts. So replace the return line with '`return listTextsPage(req, res);`' and save. Because we are internally using `nodemon https://nodemon.io/` to start our app (this is specified in '`package.json`'), the app is restarted whenever we save a file, which you can go back to your first terminal and verify.

If you think it was cumbersome to edit the file like this, then any modern IDE ought to support opening a file over a network protocol. In Emacs, for example, you open the file as usual but with the search path `/vagrant:version1--app:/home/vagrant/Version1/QFStandalone/src/index.js` , i.e. using the protocol `/vagrant:` , open a file on the machine `app` as specified in the Vagrantfile in the directory `version1` , and then the full path to the file inside the guest machine.

Another approach (which is maybe more likely) is that you would develop locally in a subdirectory to where the Vagrantfile is located. By default Vagrant mounts this directory to `/vagrant` on the guest machine (Please, do log in and check this). This is however not always the case. In some situations Vagrant has troubles keeping the host and the guest in sync, and you will be forced to sync files manually.

Summary We have now

1. Configured a virtual machine using Vagrant and a hypervisor of your choice.
2. Provisioned the machine with the help of some shell scripting so that it has node.js installed and is running a MongoDB database.
3. Installed version 1 of the QuoteFinder app.
4. Started the machine and let it set everything up.
5. Started the QuoteFinder app and tested it to ensure that it is able to connect to the web client and the database.
6. Edited a file in the QuoteFinder app and seen the app restart and the changes immediately go live.

Not bad for a first try, eh?

Cleanup Run `vagrant destroy -f` to take down the machine and scrub it from your computer. All that remains now is the Vagrantfile (Sadly, the edit we made is also lost because we never committed it), but that is also all we need to start the machine and re-provision it again.

2. Docker

The next step is to microservice the app. QFStandalone is still simple enough so that it will be run as a single microservice. However, we want to run the database as a separate microservice. If nothing else, this will mean that any books that we add to the database will remain when we rebuild the QFStandalone image.

This time, we are going to fetch the QuoteFinder app and pretend to do our development locally, so that we are only *deploying* as a microservice.

Tasks 0.[@0] Open up a terminal as before, create a new sub-directory e.g. `QuoteFinder/Version1/docker` , and cd into it.

1. Clone the QuoteFinder repository: `git clone https://github.com/mickesv/QuoteFinder/` , extract Version1 by copying it `cp -r QuoteFinder/Containers/Version1 .` , and cd into the directory `cd Version1/QFStandalone`
2. In this directory you will find a `Dockerfile` , with instructions for docker on how to create a docker image from this directory (see also Section 4.3 [Dockerfile], page 7,).

```
FROM node:18-alpine
RUN npm install -g nodemon
EXPOSE 3000
WORKDIR /app
COPY . .
RUN npm install
ENV DEBUG='qfapp:*'
ENTRYPOINT ["npm", "run", "dev"]
```

Please take a moment to study this file and make sure you understand what each line does. Pay particular notice to the line `WORKDIR /app` and the `COPY . .` instructions. With these two instructions we set the working directory inside the image to `/app` and copy everything from the current working directory on the host into the image's working directory. After these two instructions, everything that the image requires to get going is essentially in place, and all that is needed is to install dependencies from the just now inserted `package.json`.

3. Build the image: `docker build -t qfstandalone .`
4. Check that the image is indeed created: `docker image ls`
5. For the sake of it, let's tag it as well: `docker tag qfstandalone:latest mickesv/qfstandalone:version1` (replace 'mickesv' with your username on Docker

Hub). If you have an account on Docker Hub, you might also wish to push the image thither. Currently this has little meaning unless you wish to proudly distribute your own copy of QuoteFinder.

At this stage we can actually start the container, but it will not yet be able to connect to your database (you don't even have a database image installed yet). So we'll continue setting up things a while longer before we actually launch the app:

6. Download the '`mongo`' image: `docker pull mongo`
7. Create a virtual network, so that we might connect our two microservices: `docker network create qfstandalone-net`
8. Start the MongoDB container. This incantation has a few commands built in, so let's break it down first:

```
docker run                  # Start a Container
-d                          # In detached mode (in the background)
--network qfstandalone-net  # Connect to the virtual network we just created
--network-alias textstore   # Make this container accessible on the network using t
--name textstore            # Use this name when we access the container with docke
mongo                       # Use this image as base for the container
```

In one line: `docker run -d --network qfstandalone-net --network-alias textstore --name textstore mongo`

9. Now we are finally ready to start the QuoteFinder container. Again, let's break down the incantation first:

```
docker run                    # Start a container
-it                           # In interactive mode, and attach a terminal so we ca
--network qfstandalone-net    # Same virtual network
-e TEXTSTORE_HOST=textstore   # Set the environment variable to the network alias o
-w /app                       # Set the working directory inside the container
-v ./src:/app/src             # Attach the host directory ./src to the guest under
--name qfstandalone           # Docker name
-p 8080:3000                  # Connect host port 8080 to port 3000 in the containe
mickesv/qfstandalone:version1 # Use this image (the tag we previously set)
```

One long string: `docker run -it --network qfstandalone-net -e TEXTSTORE_HOST=textstore -w /app -v ./src:/app/src --name qfstandalone -p 8080:3000 mickesv/qfstandalone:version1`

10. We can now test the app. Open a web browser to `http://localhost:8080/add` and click the 'Add' button to add a book to your database. Your terminal window where the app is running will print some info, hopefully ending with *Text added.* If not, make note of any error messages printed.
11. Go to `http://localhost:8080/` and search for something, e.g. 'prince'. This will give a list of hits. Remember to keep an eye on the output in the terminal.
12. Let us now test editing the app. On your host machine, open up the file `src/index.js` , find the function '`startPage()`', replace the return line with '`return listTextsPage(req, res);`' and save. Notice that the app is reloaded directly, and when you reload the start page in your web browser you will now see a list of available texts.

The reason why this works so much more seamlessly compared to the full-virtual-machine approach we tried before is because we are mounting the host directory into the guest with the `-v ./src:/app/src` flag. In effect, we are replacing the entire '`src`' directory inside the container with whatever we have saved locally, thus overloading whatever we put in there when we created the image. If we remove the '`-v`' flag, we will revert back to whatever was

in there when we created the image. As before, `nodemon` restarts for us if it detects that a file has been changed.

If you look in the '`Version1/QFStandalone`' directory on your host computer, you will notice that it only contains the code that you yourself have created (or, <ahem!>, yours truly :-) . There is no directory `node_modules` , and there is no `package-lock.json`. These are created *inside* the container, and actually already inside the image. This is nice, you do not clutter your own disk with lots of node.js cruft (you actually do not even need to have node.js installed locally), but it does mean that you need to rebuild the image when you want to add a package dependency to your `package.json`. This is a slight hassle, but as a consolation it does not happen very often once you have settled on an architecture for your system.

13. Are we still running? Check with `docker ps` to see what containers are up and running.

Summary We have now:

1. Cloned the QuoteFinder application to our local machine
2. Created a Dockerfile and a Docker image for Version1/QFStandalone.
3. Downloaded a MongoDB docker image
4. Created a virtual network
5. Started a MongoDB database as a microservice
6. Started Version1/QFStandalone as a microservice
7. Connected Version1/QFStandalone with MongoDB using our virtual network and started the app so that we may test it.
8. Edited a file locally and see the running app inside the container restart to reflect the updated version.

Not bad for a *second* try, eh?

Cleanup

```
docker rm -f textstore qfstandalone
docker network rm qfstandalone-net
docker network prune -f
```

Remaining Pain Points

```
 ---------------------------------------
( If only there was a way to create the )
( virtual network and connect the       )
( microservices with a single command   )
( rather than having to memorise arcane )
( command line parameters...            )
 ---------------------------------------
        o   ^__^
         o  (oo)\_______
            (__)\       )\/\
                ||----w |
                ||     ||
```

6.4 A Minimal Qemu Vagrantfile

In case you want to run Qemu, here is a minimal Vagrantfile for you to start from. If you use e.g. Virtualbox, you will not need this.

```
Vagrant.configure("2") do |config|
  # Some of these things may not be necessary, but it does yield a smoother setup.
```

```
  config.vm.synced_folder ".", "/vagrant", type: "rsync"
  config.vm.boot_timeout = 999999
  config.ssh.insert_key = false
  config.vm.box_check_update = false

  # This is the most important section where you define the parameters for qemu/libvirt
  config.vm.provider :libvirt do |libvirt|
    # Don't forget to create your storage pool
    libvirt.storage_pool_name="default"
    libvirt.driver="kvm"
    libvirt.uri="qemu:///system"
    libvirt.memory = 1024
    libvirt.graphics_type = "none"
    libvirt.cpus = 1
  end

  # And this is just your bog standard Vagrantfile to fire up one virtual machine
  # using bento/ubuntu-18.04 and the name "test_machine"
  config.vm.box = "bento/ubuntu-18.04"
  config.vm.define "test_machine" do |node|
  end
end
```

7 Quiz: Let's get Practical

Please answer the following questions:

1. QFStandalone depends on five javascript packages. In what file are these dependencies specified?
2. The dependencies are never installed on your host computer. How do you instruct Docker that they should be installed?
3. How do you instruct Vagrant that they should be installed?
4. Which file/javascript module is responsible for setting up the http routes?
5. What are the responsibilities of the '`SimpleTextManager`' class?
6. What are the responsibilities of the file '`textStore.js`' ?
7. The function '`index.js::searchTexts()`' is mostly built up around promises. Where is the first Promise created?
8. Briefly outline what you would need to do if you wish to add a dependency to a new javascript package in your Vagrant solution.
9. Briefly outline what you would need to do if you wish to add a dependency to a new javascript package in your Docker solution.

There is a Quiz on Canvas where you can submit your article summaries and answers to the questions.

NOTICE: *This quiz does not contribute to the grade in the course. We do, however, require of you to submit the quiz on time.* The purpose of this quiz is to serve as a learning aid allowing you to think about these concepts, and for us to keep track of your progress in the course. If you are unable to maintain the study pace required to submit this quiz on time, we want to be made aware of this so that you are able to re-plan your commitment for the remainder of the course.

8 Provisioning and Orchestration

8.1 Provisioning

Whether we are setting up a physical computer, a virtual machine, or a microservice, we need to configure it and decide which software should be installed. This process is called *Provisioning*. When we are installing a computer or a full virtual machine, it is possible to do this by logging on to the machine and installing software interactively. The downside to this is that we create a fragile system where any particular installation may break the system in new and creative ways. When this happens, our only available solution may be to tear down the entire machine and re-install it from scratch again.

As good engineers we of course maintain a logbook of everything we do to the machine, in what order, and any errors or warnings that we get during the installation process so that we may go back and analyse what went wrong and try a different approach next time.

As better engineers, we encode our installation into a *script* that the computer can run for us. Eventually we are able to review the log output to see what went wrong and how to modify the installation script.

We have already seen two forms of provisioning; the one we made with Docker, and the one we made with Vagrant. Let us examine the differences.

With Docker we started with an almost ready machine (‘`FROM node:18-alpine`’) , and the focus was to install *anything else* that we needed and to further configure the microservice image. We also had to repeat some of this configuration on the command line when we started a container based on this image.

With Vagrant we started with a relatively clean linux install (`config.vm.box="bento/ubuntu-20.04"`), and then had to execute a series of shell commands in order to install the rest. We can not compare with the docker install straight off since we also installed a MongoDB database, and we downloaded the entire QuoteFinder repository (not just its dependencies) inside the virtual machine, but if you go back and look, you will notice that there was a bit of `sudo apt-get update`, `sudo apt-get install -y`, some `echo` and `cd here/and/there` , and some copying of files with ‘`cp`’ . All the things you could expect to do interactively, but saved in the Vagrantfile to be able to run and re-run the provisioning at will. Technically, we *can* do all of this in our Dockerfile too, but there is generally less need for it.

If we do not wish to get our hands dirty with shell provisioning, we can use provisioning software instead. For example, Ansible `https://docs.ansible.com/ansible/latest/getting_started/index.html` specifies the machines we have and their respective desired state in a collection of YAML-files, whereas Chef `https://www.chef.io/` and Puppet `https://www.puppet.com/` use their own homegrown language. While there are important differences (e.g. whether new configurations are *pushed* to each machine or *pulled* from a central repository, and whether the tools are open source or not), the overall idea is to specify a desired state for each machine rather than specific shell instructions (potentially only valid for a specific linux dialect) for how to get to that state.

An added benefit to provisioning tools like this is that it is possible to update the desired state and let the provisioning tools transfer your deployment to the new state in a controlled manner.

8.2 Orchestration

As the number of services we wish to deploy grows (actually, as soon as it grows to more than one, e.g. an application and a database), there is a need to manage the collection of

deployments. An application may, for example, consist of several collaborating virtual machines, each requiring a specific provisioning and configuration of ports to forward, or there may be a desire to create more instances of the database and have some form of load balancing between them. This additional step is called *Orchestration*. Vagrant does both; orchestration of which virtual machines to start (for example, `config.vm.define "app"` creates one machine named "app"), and the provisioning of them ('`config.vm.provision "shell"`'). This is practical for smaller setups, but the desire to *separate concerns* means that it is soon desirable to delegate the provisioning to a provisioning tool such as Ansible and leave the Vagrantfile responsible only for the orchestration.

With Vagrant, adding more machines is as easy as adding additional blocks of '`config.vm.define`'. For each defined virtual machine, the network alias, preferred IP address, and forwarded ports can be configured. Additional provisioning can also be added for each machine, but I would advise against this: it is better to rely on your designated provisioning tool for *all* provisioning, and only include as little as is necessary to bootstrap this provisioning tool in your Vagrantfile. Separation of Concerns for the win!

8.3 Docker Compose

Docker's focus is on provisioning a single image, which is then used as a template to start one or more containers. As noticed, already with two containers the command line incantations to set up a virtual network and hook up the microservices to this becomes quite a mouthful. The next step up is to define all the infrastructure in a separate file, and use '`docker compose`' to compose your application of several microservices, storage volumes, networks, etc.

Docker Compose uses a YAML file to define the services, networks, volumes, etc. that should be configured and started. A Docker compose file for the previous Version 1 of the QuoteFinder app can, for example, look as follows:

```
version: "3.8"
services:
  app:
    image: qfstandalone
    ports:
      - 8080:3000
    volumes:
      - ./Containers/Version1/QFStandalone/src:/app/src
    environment:
      TEXTSTORE_HOST: textstore
  textstore:
    image: mongo
    command: --quiet --syslog
    expose:
      - "27017"
```

And is started with '`docker compose -f docker-compose-v1.yml up`'. I am not a fan of languages where white-space has a semantic meaning, but with YAML we are unfortunately forced to accept this. Going through the example above, we see that we define two services, '`app`' and '`textstore`', each based on a separate docker image (qfstandalone and mongo, respectively). Unless otherwise specified, MongoDB is quite chatty so the '`command:`' line is basically telling it to shut up and log things using the system logger instead. Do note that the network is implicitly defined, which is a relief.

8.4 Cloud Orchestration

So now you have your Vagrantfile or your docker compose-file and are able to deploy any number of machines or services locally. The next step up is to deploy to a cloud provider instead. Vagrant has rudimentary support for this since one may define several '`provider`' blocks in the Vagrantfile, and determine at startup time whither to deploy: '`vagrant up --provider <some provider>`' . Docker compose, on the contrary, does not easily support this.

What if you want to deploy to several providers in one go? What if you want to first deploy a small set of servers, and then declaratively extend this set as the load increases? Terraform `https://developer.hashicorp.com/terraform` is one example of a tool for this. If your application is constructed only with microservices, then Kubernetes `https://kubernetes.io/` is the docker-equivalent. As far as I can tell, these two tools solve the same problem but for virtual machines versus microservices. Eventually, we will foray into Kubernetes with the QuoteFinder app, but not quite yet.

Yet one more alternative worth mentioning is Salt and its SaltStack `http://saltstack.com/` . In many ways, Salt is a combination of a provisioning tool and an orchestration tool. Once a salt infrastructure is set up (with the ever so cool and hip names '`salt-master`' and '`salt-minions`'), one distributes jobs across this infrastructure. These jobs may be to query a minion for some state, or to run a specific (UNIX) command on a particular minion. This also means that you can distribute any software to the minions so you can use and re-use your salt infrastructure over and over again with new jobs. In that sense, it is closer to parallel computing platforms such as Hadoop, or even Lambda Functions on e.g Azure or Amazon Web Services.

9 Quiz: Provisioning and Orchestration

TODO Please read the following articles and resources:

1. Rahman, A., Rahman, M. R., Parnin, C., & Williams, L. (2021). Security smells in ansible and chef scripts: A replication study. ACM Transactions on Software Engineering and Methodology (TOSEM), 30(1), 1-31.
2. Taibi, D., Lenarduzzi, V., & Pahl, C. (2018). Architectural patterns for microservices: a systematic mapping study. In CLOSER 2018: Proceedings of the 8th International Conference on Cloud Computing and Services Science; Funchal, Madeira, Portugal, 19-21 March 2018. SciTePress.
3. Khazaei, H., Barna, C., Beigi-Mohammadi, N., & Litoiu, M. (2016, December). Efficiency analysis of provisioning microservices. In 2016 IEEE International conference on cloud computing technology and science (CloudCom) (pp. 261-268). IEEE.
4. Esteban Elias Romero, Carlos David Camacho, Carlos Enrique Montenegro, Óscar Esneider Acosta, Rubén González Crespo, Elvis Eduardo Gaona, and Marcelo Herrera Martínez. 2022. Integration of DevOps Practices on a Noise Monitor System with CircleCI and Terraform. ACM Trans. Manage. Inf. Syst. 13, 4, Article 36 (December 2022), 24 pages.

These articles provide different viewpoints with which to start composing your architecture solution. First, the underlying software architecture decisions shape the possible solutions. Second, all the quality requirements (not the least including security) are still important and must be considered both on the architecture level and on the implementation level. Third, fancy principles and lofty software architectures are well and good, but once you start deploying your solution to a particular cloud provider using a particular technology stack, things soon get ugly.

TODO Summarise each article/resource (no more than 1/2 page each) according to the following:

- Authors (if available/applicable) and Title of the article
- Briefly, what is the article about?
- What have they measured?
- What are their main findings?
- What can you learn from this article?

TODO Answer the following questions:

- What is provisioning?
- What is orchestration?
- Why do you want separate tools for provisioning and orchestration?
- What is Service Oriented Architecture (SOA)?
- What "is Software-As-A-Service" (SAAS)?

There is a Quiz on Canvas where you can submit your article summaries and answers to the questions.

NOTICE: *This quiz does not contribute to the grade in the course. We do, however, require of you to submit the quiz on time.* The purpose of this quiz is to serve as a learning aid allowing you to think about these concepts, and for us to keep track of your progress in the course. If you are unable to maintain the study pace required to submit this quiz on time, we want to be made aware of this so that you are able to re-plan your commitment for the remainder of the course.

10 Application Design and Development

10.1 Principles of Microservice Architectures

The key quality requirement for cloud architecture patterns is *Scalability*. This is a bit unusual, since most modularisation efforts rather focus on *Maintainability*. Digging deeper, the goal is of course always to create a maintainably scalable solution, i.e. where scalability is built in and therefore any maintainability task that deals with scaling the application should be easy to do.

Trivially, an application can scale in two directions, i.e. *vertically* by adding more resources within each node (e.g. faster CPU's, more memory, bigger disks), or *horizontally* by adding more nodes. One refers to this as *vertically scaling up*, and *horizontally scaling out*. The goals of both these types of scalability is to support more concurrent users or jobs, or to decrease response times.

Using the same scalability mechanisms, one can also increase *reliability* of an application, e.g. by adding redundant nodes and a load balancer. Moreover, and this becomes more prevalent in microservice architectures, we can create an *extensible* architecture where it is easy to add new functionality as standalone nodes in a microservice network.

B Wilder, *Cloud Architecture Patterns*, O'Reilly, 2012. ISBN: 978-1-449-31977-9, defines some boundaries for cloud applications, that define the types of solutions one is able to imagine. Specifically, they enumerate:

(The illusion of) Infinite Resources
: Meaning, horizontal scaling is easy and preferred. It also means that there is no upper limit to how many services you can use to deploy in your application.

Elastic Scaling
: When more resources are needed, they are available, but equally important is that you are able to release resources when they are no longer needed.

Metered Billing
: This is the term that Rosenberg & Mateos, *The cloud at your service*. Manning Publications Co. 2010. uses. You only pay for the resources that are currently being used. Again, the idea is to lull you into thinking that resources are infinite and infinitely cheap.

Automation
: This scaling up and down is managed through automated tools, where the currently desired platform is programmatically expressed (Infrastructure as Code). There is no need for manual intervention by a platform engineer e.g. to make new resources available, or for new contracts to be negotiated for every change in the required cloud resources.

As a developer of a cloud-native application, these principles are in fact more or less sufficient. The knowledge that new resources are always available and under the control of the application itself, as and when needed means that an application can be developed as a collection of stand-alone units. For the *cloud provider*, however, a few more principles are desirable to make this an economically viable business model:

Multitenancy, or Pooled Resources
: There may be any number of virtual resources running on each physical hardware unit, and any single user of a cloud service or of cloud infrastructure is not aware of who else may be sharing the physical platform at any given point. Each resource has its own CPU, memory, and disk, and is not aware of what else may be running next to it (or what they are paying for).

Virtualisation
: In order to support the above, virtualisation of resources is a required technical solution. Without virtualisation, neither multitenancy, automation, nor elastic scaling, would be possible.

Commodity Hardware
: To further keep the costs down for the cloud provider, it is desirable to avoid any specialised hardware. For example the CPUs, memory units, and hard drives should be as close to off-the-shelf units as possible, that could just as well be installed in a desktop computer.

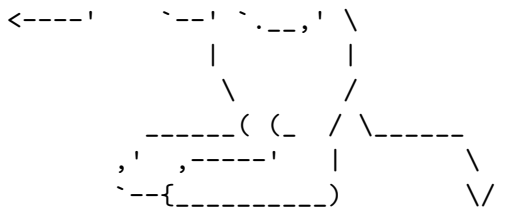
Where the first group of principles *enabled* cloud native applications, these additional principles *restrict* the solutions. Critically, a cloud application can never be sure that a specific resource is available at a specific point in time. It might, for example, be in the process of migrating to a new hardware unit to make room for another customer's request for a bigger resource, or the hard disk may fail. As usual (to use a Swedish expression), what you gain on the carousel, you loose on the swings. The cost per time unit for a specific cloud resource may be low, but this is somewhat balanced by the cost of the development time to create a fault-tolerant application.

Platform Service Ecosystem
: This is one last piece of the puzzle for a cloud native application. A cloud solution requires more than just virtual CPU power. For example, data storage, a message bus, network facilities, monitoring facilities, the ability to move the application closer to its users, etc., may also be required for a complete cloud application. Different cloud providers may provide a more or less rich ecosystem of platform services. When selecting between different cloud providers, these extra services may and should factor in the decision.

10.2 Cloud Architecture Patterns

B Wilder, *Cloud Architecture Patterns*, O'Reilly, 2012 is a good starting point for the basic structures available to create scalable software architecture solutions for cloud native applications. Below is a brief summary of some of the patterns in this book; for details I encourage you to read the book yourself.

Horizontally Scaling Compute Pattern and the **Auto-Scaling Pattern** Together, these two patterns suggest to prefer horizontal scaling and to use automated tools to request this scaling. When using horizontal scaling, each component in the application can more or less be single-threaded and should be kept as simple as possible. As I'm sure have been hammered into you by now, any time you are forced to write multi-threaded code an angel loses their wings.



Challenges with horizontal scaling includes the need to be absolutely sure about the billing model the cloud provider uses, how to deal with session state, and how to update the nodes with a minimum of application downtime.

Queue-Centric Workflow Pattern This pattern is used to make the nodes more loosely connected. Without this pattern, when one node wishes to communicate something to another node (For example, a control message or a job request), it first needs to find the right instance of the other node, and then wait for it to become available to receive the message. Meanwhile, the first node is also unavailable for new requests.

With the queue pattern in place, rather than going directly to the other node, the message is placed in a queue, and the first node can go back to being available. When one instance of the receiving node type becomes available, it will collect the messages in turn from the message queue and act upon them. This pattern is usually not relevant for read-only requests, since it is desirable to keep these synchronous. Rather, it is used when a job can be launched asynchronously and the calling node need not wait for the job to finish.

The actual queue can be implemented in many ways, either through a message bus provided by the cloud platform, a dedicated MQTT `https://mqtt.org/` solution running on a separate node, or using some storage resource already available to all nodes (e.g. a shared filesystem or a database).

The queue should be more reliable than the nodes acting on the data so that no messages are lost, and it should be possible to re-process the same message multiple times with the same business result.

MapReduce Pattern This gives me a chance to write about my favourite programming language, *lisp*, where this pattern is an intrinsic part of any data processing. Surprisingly, it reappears in more modern programming languages as well, such as JavaScript and Java (with the introduction of the Streams API). The underlying principle of this programming pattern is to transform a list into another list by applying a function to each element in the list, i.e. a *map* from one domain to another. Multiple transformations can be daisy-chained through a series of *maps*, and often this is sufficient to process the data as required. Sometimes, something needs to be computed across an entire list, i.e. *reduce* the list down to for example a scalar. This is done by applying a function together with an accumulator variable to each element in the list.

```
              map                  reduce
(a b c)  -------> (a' b' c') ----------> X
```

As it happens, this programming pattern is eminently suitable for scaling. In the extreme case, each application of the map function can be distributed to a separate node. Since each element in the list and the application of the map function on that element is independent from all other elements, they can be executed asynchronously, and the results collected and collated into the modified list. The reduce function can be trickier to distribute since there may be a dependency on that elements are reduced in a specific order. However, with some clever use of the right data structures (in particular the *dictionary* data structure), it is possible to create parallelisable versions of reduce functions as well.

When parallelising this programming pattern e.g. in a cloud native application, important aspects to consider is to ensure that each element is truly independent of all other elements, and that the map function can operate on each element individually. Additional map/reduce steps may be necessary in order to preprocess the data to ensure this. Scalability-wise, there is an

overhead cost for a new computing node, since data need to be pre-processed and distributed to the node and the results need to be collected and collated, so it is important to fine-tune the number of jobs assigned to each computing node for optimal performance.

Node Failure and **Busy Signal** patterns The fundamental programming principle in a cloud native application is that *All application compute nodes should be ready to fail at any time.* As a corollary to this, all application compute nodes must be prepared that any other node is about to fail at any time.

Protecting from sudden and unplanned node failures, data that should be persistent should be kept in a persistent storage, jobs that have not been completed should remain in the job queue, and one might consider to have a certain number of redundant nodes that can pick up the slack when a node fails.

Nodes subjected to a planned failure (either planned by the cloud provider or by the application itself) have the luxury of both being able to complete what they are currently working on, gracefully removing themselves from accepting new jobs, and to alert e.g. load balancers that they are about to become unavailable.

From the calling side (be it the end user or another node inside your application), it is important to maintain an updated list of available nodes, or delegate this to a load balancer or a worker queue. Calls between nodes should be programmed with a timeout to enable switching to an alternative strategy, should a node fail to reply. When a call times out, the options are to retry, wait a while and then retry, or switch to a backup node and retry. The 'retry' strategy is often sufficient if there is a load balancer in place, since the new call is likely to end up on a new node anyway.

When and whether to alert the load balancer that a node might be unavailable is another discussion.

Sometimes, a node is not completely down, it is just unable to process the request right now. For example, it did not get all of the message, some other component it relies on was temporarily unavailable, or it was busy doing some maintenance work. In that case, the node should return a "busy signal", for example a '`503 service unavailable`' return on a http request. Good clients recognise this busy signal and have strategies for dealing with this. Usually, a retry is sufficient, or maybe first wait a while and then retry.

Colocate Pattern and **Multisite Deployment Pattern** Nodes that work closely together should also be located as close to each other as is possible. Later on, you will learn how tools such as Kubernetes have built this into their microservices deployment architecture.

This principle extends to the end users. By deploying an application to many data centers across the world, users can be routed to the instance running closest to them, thereby reducing network latency. As it happens, this also enables redundancy in that if one data center fails, the work can be handed over to one of the application instances running on a different data center.

There is an overhead and a cost in running multiple instances of an application – even when using the cloud provider's tools for moving the application around the world. Unless there is a tangible improvement in user experience, it may not be worth it to implement the multisite deployment pattern.

Another consideration is whether your deployments across the globe should shrink and grow over the course of a single day, to accommodate for the fact that users in a particular region are more or less active during a day (e.g. more activity in the early evenings and less at night).

Content Delivery Network (CDN) and **Valet Key** patterns Building a RESTful microservice usually means starting with some web framework in your programming environment (e.g. an express application in node.js). This means that you have an adequate or possibly decent web server that is good at executing the application code, but may not be great as a web server. If your application is serving a lot of static contents, then it may not be able to keep up with the dynamic requests that will execute your application code.

The solution is to include a dedicated web server in your solution that is really good and optimised for serving static content. Either access this web server directly, or through dedicated nodes across the globe whose purpose is to cache oft-requested contents closer to the end-user. This network is solely focused on delivering contents and not bothered with anything else in your application. Specifically, the CDN does not deal with dynamic contents.

Sometimes (oftentimes) the delivered content should only be available to the one user that requested it (or at least a limited list of users), or it should only be available for a specific period of time. That's where the Valet Key pattern comes in. When a user requests access to some content through your application, a unique URL is generated for this particular user, and the CDN server is instructed to make the requested resource available on that URL. This URL may also be set up to only work for a limited time. Moreover, the URL may be used to also upload contents instead of just downloading or streaming.

If a really secure solution is desired, the URL should only be valid for a limited time and from a specific IP address. Furthermore, the contents can be encrypted so that the user must also have access to the decryption key.

Summary The architecture patterns for cloud native applications are centered around the ability to scale up and down as needed (This is perhaps not very surprising), but the preference is to scale horizontally. Since the current state of the application (e.g. the number of available nodes of any particular type) can change at a moments notice, robust communication pathways using e.g. message queues are required. To some extent, this defines the execution view of the system's architecture, but with this definition comes a few limitations. Accepting that nodes can fail or be taken offline at any time means that applications need to be built with this in mind. Limitations in network speeds and a desire to make the best use of available computing resources influence where and how an application's services are deployed (e.g., which parts of an application that are co-located or distributed closer to the customers) and indeed what constitutes a service in the first place (e.g. inserting a content delivery network in the application). In a more conceptual focus, the desire to enable scaling implies restructuring algorithms into more scaling-friendly forms, for example the MapReduce pattern. Finally, an open architecture across multiple nodes and servers requires that security and access control is considered throughout the application, for example using temporary and time-limited access keys.

10.3 Introduction to YAML and Docker-Compose.yml

An increasing range of tools use YAML `https://yaml.org/spec/` to describe configurations. As mentioned before, this includes provisioning tools such as Ansible, and orchestration tools such as Docker Compose and Kubernetes (which we will eventually get to). There are of course other choices too, but it is a rather neat idea to be able to use a single grammar for all types of configurations. It is claimed that YAML is more readable than JSON (which I doubt) and XML (which I agree with). Below is a quick rundown of relevant YAML syntax (there is more, but this covers 90% of the cloud configuration use cases).

```
---                          # Every new YAML node starts with three hyphens. You can c
Name: "Arthur Dent"          # Something with the key "Name" has the (string) value "Ar
Age: 42                      # Integers and floats are also supported.
Active: true                 # ... as are booleans (true or false)
Friends: null                # ... and null values.
Inventory:                   # The value of the inventory key is a list.
  - Tea
  - No Tea
  - Towel
  - Babelfish
```

```
Desires: [ House, Clothes ]   # Lists can also be inlined
Address:                      # A Mapping (or a dictionary) is a set of key/value pairs
  Street: "Vogon Intergalactic Bypass #47111"
  City: "What do you mean city? We work with the whole Universe!"
  HouseNumber: 0
Contacts:                     # A list of mappings
  - Name: "Ford Prefect"
    Address: "Anywhere, really. Just wave!"
  - Name: "Trillian"
    Address: "Heart of Gold"
  - Name: "Marvin"
    Address: "Everywhere, eventually."
Appearances:                  # Inlined mappings
  - {Title: "The Hitchhiker's Guide to the Galaxy", Year: 1979}
  - {Title: "The Restaurant at the End of the Universe", Year: 1980}
  - {Title: "Life, the Universe and Everything", Year: 1982}
  - {Title: "So Long, and thanks for All the Fish", Year: 1984}
  - {Title: "Mostly Harmless", Year: 1992}
FavouritePoem: |              # Multi-line contents.
  Oh freddled gruntbuggly,
  Thy micturations are to me, (with big yawning)
  As plurdled gabbleblotchits, in midsummer morning
  On a lurgid bee,
  That mordiously hath blurted out,
  Its earted jurtles, grumbling
  Into a rancid festering confectious organ squealer. [drowned out by moaning and screa
...                           # Formally, a YAML ends with three dots. Can often be left
```

Do note that indentation matters. Apparently this is a popular design decision these days. I personally dislike this since I claim that whitespace should not carry semantic meaning. In my not so humble opinion, this choice decreases maintainability and readability.

On to Docker Compose, then, the configuration of which is specified in a yaml file (Please see `https://docs.docker.com/compose/compose-file/` for a full reference). In summary, the following can be specified:

- '`Services`' (required) correspond to your different microservices, i.e. the components with which you will build your system, and for each service you specify its configuration, and how to scale it etc.
- '`Networks`' Can be used to separate parts of your application into smaller networks. Usually, there is not much configuration to do per network.
- '`Volumes`' define persistent storage volumes. Persistent is key here; if you are only using a temporary storage inside a container, you do not need a volume, but if you want to store data that survives restarts of the container (or scaling of it), then you define a volume for this data. Typically, you will at least want volumes for use by your database service.
- '`Configs`' are used to modify the configuration inside a container without having to rebuild the image. For example, this can be done by superimposing a new configuration file over the one in the image.
- '`Secrets`' a special type of configs for sensitive data. It is good practice to keep configs and secrets separate to avoid accidentally exposing them (e.g. by committing a secret in the configuration management tool).

A brief example:

```
services:
```

```
  name-of-some-service:
    image:            # docker image to base this service on
    ports:            # should match what you wrote in your Dockerfile
      - "host:container"
      - "8080:3000"
    volumes:
      - "volume:container-path"
      - db-data:/etc/data  # Mount the volume db-data inside the container on the path
  some-other-service:
    image:           # the image of some other service you want to run

volumes:
  db-data:
```

. . . And at this level, there really isn't much more to say about the yaml syntax.

11 Let's get Practical Again

11.1 Introducing Version 2 of the QuoteFinder app

Version 2 of the QuoteFinder app replaces the rudimentary search algorithm in the first version with something more complex that is able to look for the words *near* each other and not just directly adjacent to each other, and instead of returning just a certain number of characters around where the words were found, this version returns the full sentences where the words were found.

I know. This is still not very complex and we could just as well continue to run this new search algorithm in a single service, but let us *imagine* that this is super-complex and require a not insignificant amount of time to go through a single text. Let us imagine that as the number of books in our database grows, a single user request to look for a quote will require the help of several worker nodes sifting through the database.

The first and most obvious thing to do is to define an architecture that enables horizontal scaling, and that makes use of a message queue to dispatch jobs to the worker nodes.

QFApp Frontend The first microservice, QFApp, is similar to the previous QFStandalone, in that it is an express app. Much of the code is similar, including the use of a MongoDB database to store the texts in. The main difference lies in what happens in the '`searchTexts()`' function. Rather than actually performing the search and format the resulting output, we are now invoking a '`Dispatcher`' to distribute the search jobs onto a message queue to be consumed elsewhere by a number of '`QFWorkers`'.

The message queue used by the Dispatcher is really the first good hit I found on NPM, called the Real Simple Message Queue, or '`RSMQ`'. To be precise, I prefer the promise-enabled version of this queue '`RSMQPromise`'. This queue makes use of a Redis key-value store to enqueue and dequeue messages to and from, so we need to remember to add this to our deployment architecture.

A challenge here, which should be a warning flag that this app is not really a good candidate for the queue centric architecture pattern, is that once the worker nodes are done they need a way to return what they find so that this can be returned to the web user. In this implementation the Dispatcher sets up a return queue, sends the jobs, waits for messages on the return queue, and then sends these on to the web interface using '`socket.io`' emit-calls.

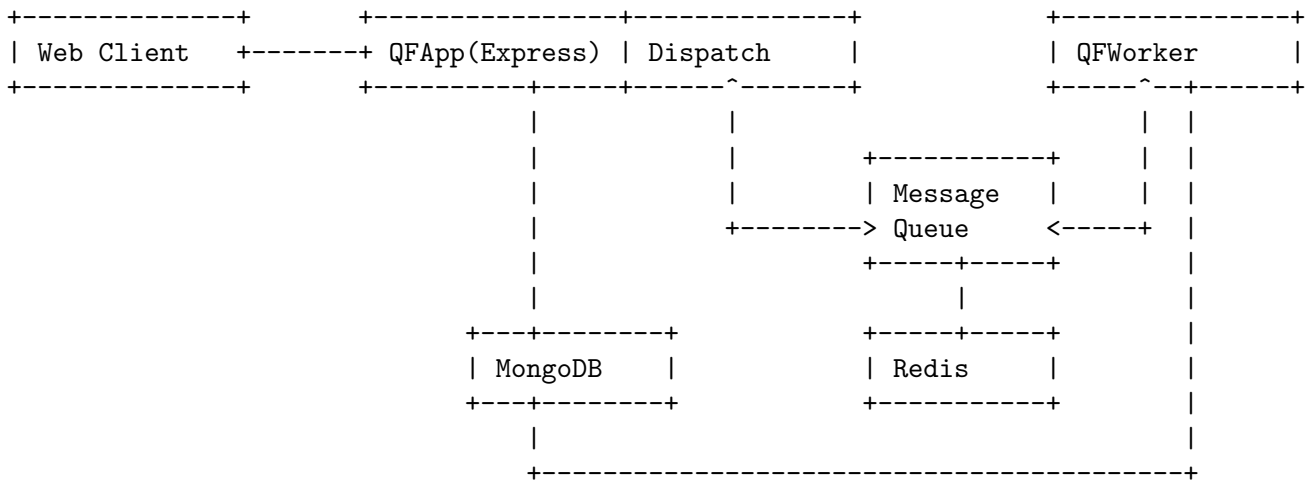
QFWorker Backend In contrast, the backend is much simpler. It connects to the job queue and listens for messages. The first one to pick a message gets it, and will go to town trying to find all the quotes. Once found, these are sent on the return queue, and eventually the QFWorker will either pick the next message if there is one, or go to sleep waiting for a new job.

To make things more interesting and to continue imagining that this is a complex and resource intensive application, I have added a bit of vertical scaling in here as well: Every job consists of a single title in the database, and this title is split up into chunks (of '`CHUNKSIZE=1000`' lines), and a number of chunks ('`MAXTHREADS=10`') are searched in parallel.

Data Format One reason for this decision is that I want you to get used to thinking about your data architecture as well as your application and deployment architectures.

Consider this: Each text is added to the database once, but is searched multiple times. If we keep the text as a single block in the database but still want to parallelise the search algorithm, then for every query we need to (a) fetch each entire text at once, and (b) split it into the desired chunk size. If we instead do this when we first add the text to the database, each search is made simpler because it only needs to fetch as many pre-split chunks as it currently need from the database, and then work on these. Less work for each search and less data to fetch before starting to analyse implies that we can start returning results sooner.

Summary A conceptual view of QuoteFinder Version 2:



The overall process for a search:

Web Client

1. search(words)
2. Listen on '`socket`' for results

17. Display results.
18. Goto 2.

QFApp

3. List all texts in TextStore database
4. Create a job for each text
5. Create a return queue
6. Send all jobs to the 'searchJob' queue
7. Listen on the return queue.

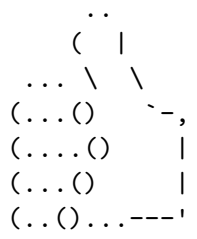
14. Receive results on return queue.
15. send to web client's '`socket`'
16. Goto 7.

QFWorker

0. Listen on 'searc

8. Pick first messa
9. Create N thread
10. [each thread] f
11. [each thread] l
12. Collate and se
13. Goto 9 until n
then goto 8 until n
then goto 0.

Simple, isn't it?



11.2 Installing and Running Version 2 of QuoteFinder

You have already installed the required software stack (`docker` and `docker-compose`), so this time we can get started directly with defining the deployment architecture.

Tasks

0. Open up a terminal and enter your `QuoteFinder` directory.
1. Clone the QuoteFinder repository: `git clone https://github.com/mickesv/QuoteFinder/` (If you did this in the previous practice round, git will throw an error but that's ok) extract Version 2 by copying it `cp -r QuoteFinder/Containers/Version2 .` , and cd into the directory `cd Version2`
2. This time you will find two sub-directories here, one for '`QFApp`' and one for '`QFWorker`'. As before, inside these directories you will find a Dockerfile. Please take a moment to study these two files. You will notice that they are almost the same. In fact, the only differences are that QFApp will expose port 3000, and that the '`DEBUG`' environment variable is different.
3. Build the two images with the tag-names '`qfapp`' and '`qfworker`', respectively.
4. MongoDB is already downloaded, but you need to pull '`redis:alpine`' as well.
5. Back up to the directory `QuoteFinder/Version2/` and Create a file '`docker-compose-v2.yml`'.

In this file, you want to define the following '`services`':

- '`app`' is based on the '`qfapp`' image
  - make sure '`ports`' 8080:3000 are forwarded (port 3000 on the guest is forwarded to port 8080 on the host)
  - the following '`environment`' variables are set: '`REDIS_HOST: messagequeue`' '`TEXTSTORE_HOST: textstore`'
- '`worker`' is based on the '`qfworker`' image
  - the following '`environment`' variables are set: '`REDIS_HOST: messagequeue`' '`TEXTSTORE_HOST: textstore`'
- '`messagequeue`' is based on the image '`redis:alpine`'
  - its starting '`command`' is '`redis-server`'
  - it '`expose`' the port "6379"
  - it mounts the following '`volumes`'
    - '`redis-data:/data`'
    - '`redis-conf:/usr/local/etc/redis/redis.conf`'
- '`textstore`' is based on the image '`mongo`'
  - its '`command`' is '`--quiet --syslog`'
  - it '`expose`' the port "27017"
  - it mounts the following '`volumes`'
    - '`textstore-data:/data/db`'
    - '`mongo-config:/data/configdb`'

The volumes used need to be defined (it is, however, enough to mention them, no further configuration is necessary):

```
volumes:
  redis-data:
  redis-conf:
  textstore-data:
  mongo-config:
```

Note that you are intentionally left to explore the docker-compose syntax on your own here. Play around and give it a try before cheating. And no, I'm not going to tell you how to cheat.

7. Run `docker compose -f docker-compose-v2.yml up` to start your setup.

8. Open a web browser to `http://localhost:8080/add` and click the 'Add' button to add a book to your database. Your terminal window where the app is running will print some info, hopefully ending with *Text added.* If not, make note of any error messages printed.
9. Go to `http://localhost:8080/` and search for something, e.g. 'prince'. This will give a list of hits. Remember to keep an eye on the output in the terminal.

11.3 Editing the Running Application

In case you happened to skip directly to the search step, you will notice that the search fails on a timeout and the worker node does not seem to be functioning at all. Let us see if we can recreate this bug. In the console, press '`Ctrl-C`' to stop the running deployment, then re-run the same command as before but go directly to the search page `http://localhost:8080/` and repeat the previous search. If you had already added something, it is still there and you can not reproduce the bug. Please take a moment to consider why this is the case.

Tasks

1. Inspect the running deployment. Open another terminal and run `docker ps`. You should see four containers running. You might also wish to see what volumes are available: `docker volume ls`. Among other volumes you should see the four volumes defined in your yml-file. We can not remove them directly because they are in use by running containers (try! The command is `docker volume rm <volume-name>`). Stopping the deployment doesn't help either, the volumes are apparently still in use.
2. Stop the deployment with '`Ctrl-C`' and run `docker ps`. As you notice, the list is empty. You need to look for *all*, even exited containers, with the command `docker ps -a` to see them. Let us remove all the containers running or not. I will do this in a lazy way with the command `docker rm -f $(docker ps -a -q) dummy` (the nested command lists only the IDs of the containers, and the outer command removes them. 'dummy' is appended just to ensure that there is something – anything – there to look for so the outer command knows to behave nicely).
3. It should now be possible to remove the volumes. Please do, and then restart the deployment again. Visit the search page `http://localhost:8080/` , search for something, and see it timeout and fail.
1. Assume that we now want to edit the application, as we did before. With its current deployment, this will not work; each container is only based on the initial image (except redis and mongo that use the defined volumes). In order to enable editing we must – as before – mount a directory on the host computer to a directory inside the container (this is referred to as a *bind mount* in docker vernacular). Edit the `docker-compose-v2.yml` file and add `volumes` to the services `app` (`./QFApp/src:/app/src`) and `worker` (`./QFWorker/src:/app/src`).
2. Restart the deployment (technically, you only need to re-read the configuration, but for now it is easy enough to just '`Ctrl-C Up Enter`' to restart).
3. On your host machine, open up the file `QFApp/src/dispatcher.js` and familiarise yourself with the code. In the method `formatJobs()` we want to add the following at the start of the function:

```
if (0 == texts.length) {
    console.log('WARNING: No texts are available, no jobs will be created.');
}
```

As before, you should see the app reload directly (in the console output), and if you search for something in the web browser before adding any texts this new warning will be printed in the console.

11.4 Scaling the Deployment

There is one last thing we wish to do with the running deployment: Scale it. With `docker compose` this is in fact quite easy:

Tasks

1. Edit the file `docker-compose-v2.yml` and add the following instruction to the `worker` service:

   ```
   deploy:
     replicas: 3
   ```

2. in a terminal where you are not running the `docker compose ... up` command, enter the directory where the file `docker-compose-v2.yml` is located, and run the command `docker compose -f docker-compose-v2.yml up -d` (Note the 'detach' flag `-d`, this is what does all the magic in this case). Look at the output in the first terminal, you should see that there are now three replicas of the worker node running.
3. Edit the number of replicas down to 1 again, re-run the `docker compose .. up -d` command and see what happens with the running deployment.

In theory, it is possible to scale the `app` service in the same way. In practice, however, this will not work. The reason for this is that the web client uses `socket.io` for communication, which essentially ties one web client to one running app container. The built-in load balancer in docker compose employs a simple round-robin algorithm, passing each new call to localhost:8080 to port 3000 in the next `app` container.

11.5 Cleanup and Summary

Stop and clean up after the running deployment by running the following commands:

```
docker compose -f docker-compose-v2.yml stop
docker rm -f $(docker ps -a -q) dummy
docker volume rm version2_mongo-config version2_redis-conf version2_redis-data version2
docker network prune -f
```

The last command may need a clarification: Even if we have not manually created any virtual networks, docker compose have still created them in the background. `docker network ls` will show you any remaining virtual networks.

Summary We have now:

1. Built two Docker images, corresponding to the frontend `qfapp` and the backend `qfworker` of the QuoteFinder version 2 app.
2. Downloaded a '`redis:alpine`' image.
3. Constructed a docker-compose configuration that defines the four services and the volumes needed for this version to run.
4. Launched this deployment
5. Modified the configuration to use *build mounts* to superimpose the host's version of the application code over the default code inside the images.
6. Edited a file locally and see the app in the running deployment restart to reflect the updated version.
7. Modified the configuration to add more *replicas* of the `worker` service.
8. Re-deployed the configuration to the running deployment, without a restart.

Please take a moment to compare these achievements with the manual deployment of QuoteFinder Version 1.

Remaining Pain Points Using RSMQ for passing messages between the frontend and the backend turned out to be a bad idea for a number of reasons:

1. The implementation of RMSQWorker is in fact *polling* the Redis store, so there is an inevitable delay before jobs are picked up.
2. That we create a unique return queue and pass that along in the job is a dependency injection, meaning that the containers are no longer independent of each other.
3. The workers need to pass their output back to a waiting web client. This means that the queue centric workflow is not the most suitable design pattern.

A more apt design pattern is to use a REST-API to access the QFWorkers; this is explored in Version 3 of the QuoteFinder app.

Application design aside, the use of docker compose poses a bigger problem: Docker Compose deploys locally and not to a cloud provider. Moreover, there is no easy mechanism to distribute containers across several hosts.

We can scale the QFWorker by adding more replicas, but we cannot easily scale the frontend QFApp. Making the frontend simpler, acting as a gateway to the QFWorkers who do the heavy lifting, is one approach to deal with this.

```
 ----------------------------
< Can we scale the database? >
 ----------------------------
        \   ^__^
         \  (oo)\_______
            (__)\       )\/\
                ||----w |
                ||     ||
```

11.6 Introducing Version 3 of the QuoteFinder app

A few issues continue to itch with QuoteFinder Version 2. Specifically, the use of a queue centric workflow for an application where this is not the most suitable choice, and the resulting bad performance due to the chosen queue implementation. Even more concerning is that the wrong choice of architecture pattern broke one of the fundamental principles of microservice design: *Services should be independent*! Before continuing, let us take a moment to correct this.

In `https://github.com/mickesv/QuoteFinder/` , that you have previously cloned, there is also a Version 3 of the QuoteFinder app. Please copy this version to a separate directory:

```
cp -r QuoteFinder/Containers/Version3 .
cp QuoteFinder/docker-compose-v3.yml Version3/
cd Version3
ls
```

As before, you should see two directories, one each for `QFApp` and one for `QFWorker`. You also copied a file `docker-compose-v3.yml`, so let's start by inspecting this file:

```
version: "3.8"
services:
  app:
    image: qfappv3
    ports:
      - 8080:3000
    volumes:
      - ./Containers/Version3/QFApp/src:/app/src
    environment:
```

```
      TEXTSTORE_HOST: textstore
      WORKER: worker
      TIMEOUT: 10

  worker:
    image: qfworkerv3
    volumes:
      - ./Containers/Version3/QFWorker/src:/app/src
    deploy:
      replicas: 5
    environment:
      TEXTSTORE_HOST: textstore

  textstore:
    image: mongo
    restart: always
    command: --quiet --syslog
    expose:
      - "27017"
    volumes:
      - textstore-data:/data/db
      - mongo-config:/data/configdb

volumes:
  textstore-data:
  mongo-config:
```

There are only three services this time since the Redis store is no longer needed. The images are tagged `v3` to keep them separate from the previous version (They could probably have been named `qfapp:v3` and `qfworker:v3` according to Docker's tag-numbering scheme, but they are not really versions of the same app since they use a different architecture, and I like to live on the wild side). The `app` service has one additional environment variable set, but otherwise there are no major differences.

Inspecting the QFAapp frontend, the only part we expect to differ is the `Dispatch` class (This can be confirmed with the help of the command `diff`, for example using `diff --brief QFApp/src/ ../Version2/QFApp/src/` to get a summary of which files that differ). In the same way, we find that the only file that differs for the QFWorker is the `index.js` file.

The `Dispatch` class is now much simpler, since there is no need to meddle with any queues in any direction. Since it is now so small, let us inspect the class in its entirety:

```
class Dispatcher {
    constructor() {
        this.TIMEOUT = 1000 * (process.env.TIMEOUT || DEFAULTTIMEOUT);
        this.WORKER = process.env.WORKER || DEFAULTWORKER;
    }

    formatJobs(searchString, texts) {
        return texts.map( t => { return {searchString: searchString, textTitle: t};});
    }

    dispatchSearch(searchString, jobs, socket) {
        console.log('Searching for : ' + searchString);
        let baseurl = 'http://' + this.WORKER + ':3000';
```

```
            if (0 >= jobs.length) {
                socket.emit('done', {msg: 'no texts available' });
                return 'DONE'
            };

            setTimeout( () => socket.emit('done', {msg: 'timeout'}), this.TIMEOUT);

            return Promise.all(jobs.map( j => {
                let title = j.textTitle.replaceAll(' ','+');
                let search = j.searchString.replaceAll(' ','+');
                let url = baseurl + '/' + title + '/' +search;
                console.log('Using url:', url);
                return fetch(url)
                    .then(res => res.json())
                    .then(res => res.forEach( t => socket.emit('answer', JSON.stringify(t)
            }))
            .then( () => socket.emit('done', {msg: 'done'}));
        }
    }
```

As before, `formatJobs()` generates one job for each text in the database (and could be improved with the same modification we did for Version 2). The new magic start with the line `return Promise.all(jobs.map( j => {` , where each job is used to generate a Promise to dispatch the search of that specific title to one worker. Each title will generate an asynchronous call to the REST API in a QFWorker (the load balancer in Docker Compose decides which one), and when an answer is received it will use `socket.emit()` to send this answer to the web client.

The `index.js` in QFWorker is now slightly longer since it is now using the same '`express.js`' framework as QFApp to serve up a tiny web server. However, most of the new code is boilerplate code for '`express.js`', and there is really only one method which takes care of the search requests. Please take a moment to study `index.js` and make sure you understand how the REST API is constructed.

Feel free to build the necessary images for Version 3, start the deployment and play around with it. It should be prepared for you so that you can live-edit the files and the deployment as you did with Version 2.

12 Quiz: Design Decisions and Development Setup

TODO Please read the following article:

- Taibi, D., Lenarduzzi, V., & Pahl, C. (2018). Architectural patterns for microservices: a systematic mapping study. In CLOSER 2018: Proceedings of the 8th International Conference on Cloud Computing and Services Science, SciTePress, Setúbal, 2018.

TODO Answer the following questions on Taibi et al.:

- Briefly list and describe the microservice architectural patterns identified in Taibi et al.?
- How do these patterns differ from the architecture patterns described by Bill Wilder, *Cloud Architecture Patterns*, O'Reilly, 2012 (You can find a summary in Section 10.2 [Cloud Architecture Patterns], page 30)? Are there any similarities?
- Taibi et al. describe a number of guiding principles for microservice architecture styles. Please list and briefly describe each of these.

TODO Answer the following questions:

- What patterns (from Taibi et al. as well as from Wilder) are currently used in QuoteFinder Version 2?
- What are the bottlenecks in the current architecture? What can we do about them?
- What can be done to scale QuoteFinder Version 2? Think in "both ends", i.e. scaling the front-end as well as the back-end.

There is a Quiz on Canvas where you can submit your article summaries and answers to the questions.

NOTICE: *This quiz does not contribute to the grade in the course. We do, however, require of you to submit the quiz on time.* The purpose of this quiz is to serve as a learning aid allowing you to think about these concepts, and for us to keep track of your progress in the course. If you are unable to maintain the study pace required to submit this quiz on time, we want to be made aware of this so that you are able to re-plan your commitment for the remainder of the course.

13 Deployment with Kubernetes

13.1 Introduction to Kubernetes

With the introduction of Docker Compose, it is now possible to provision and orchestrate docker microservices in the same way as has previously been introduced for full virtual machines e.g. with the use of Vagrant. Some differences are nevertheless worth mentioning:

With Vagrant and a hypervisor, provisioning and deployment is done in one step. While a separate provisioning tool such as Ansible or Puppet may be used, the idea is still to start with an empty virtual machine and then install and configure everything from the operating system and required tool stack all the way up to the software that is supposed to run on the VM.

With Docker the provisioning is done in a separate step, where an initial docker image is created. Up to a point, the software in this image is ready to start running, requiring only a small set of additional run-time configuration. Docker compose is responsible for the next step, which is to orchestrate and deploy the specified services.

There is, however, one more trick up Vagrant's sleeve, and that is the ability to deploy to different providers. Adding a small configuration block in the Vagrantfile makes it possible to use the same configuration to deploy to many different cloud providers (or, should you so desire, different hypervisors locally).

This is not supported by Docker Compose. To this end, it is time to introduce the next tool in the tool chain, i.e. Kubernetes `https://kubernetes.io/` . Somewhat simplified, one can say that Kubernetes is used to set up the deployment infrastructure within which the application microservices are run.

Some Kubernetes Terminology From the inside out:

1. *Containers* are still the core concept, and they are no different to the microservice containers used previously. They are, in fact, based on the same docker images as before.
2. Containers are collected into *pods*, that gathers containers working closely together, possibly with some shared storage.
3. A container can mount *Volumes* inside the same pod.
4. If the container need *Persistent Storage* this is split into two parts. The developer of container makes a *PersistentVolumeClaim* describing what type of persistent storage they need, and the administrator of the Kubernetes cluster provides one or more *PersistentVolume* to match the desired claims.
5. A *Deployment* describes the contents in a Pod on a slightly higher level, enabling Kubernetes to take more responsibility for e.g. scaling the deployment or to move the pods to other nodes if necessary.
6. *Labels* are used to identify which parts of the application are affected by different configuration directives.
7. *Services* are set up to access your application. They act as the glue between the pods/containers and the outside world.

The best way to come to terms with this is to walk through a practical example, which is what we'll do next.

13.2 Get Started with Kubernetes

You have already installed the necessary software to set up a very small Kubernetes cluster (consisting of a single machine). For Windows/Mac, this is included in `Docker Desktop` , although you may need to go into the settings and enable Kubernetes. For Linux, you will use the

standalone `minikube`, started with a simple `minikube start` . Check the status of your cluster with `kubectl cluster-info` .

It is possible to set up your deployment well and good using only the command line tool `kubectl` but I would not recommend it. By now, the mantra *Infrastructure as Code* should be ingrained into you, so let's write some YAML code to set up a deployment.

```
# QFStandalone
# --------------------
# - Deployment to launch one container of mickesv/qfstandalone in a pod.
# - Service (type: LoadBalancer) to open up the app to the world (localhost, at least)
#
---
apiVersion: v1
kind: Service
metadata:
  name: qfapp-service
  labels:
    app: qfapp
spec:
  type: LoadBalancer
  ports:
  - name: http
    port: 3000
    nodePort: 30001
    protocol: TCP
  selector:
    app: qfapp
---
apiVersion: apps/v1
kind: Deployment
metadata:
  name: qfstandalone
  labels:
    app: qfapp
spec:
  replicas: 1
  selector:
    matchLabels:
      app: qfapp
  template:
    metadata:
      labels:
        app: qfapp
    spec:
      containers:
      - name: qfstandalone
        image: mickesv/qfstandalone:version1
        ports:
        - containerPort: 3000
---
```

A quick summary of what can be seen:

- This consist of two yaml blocks (separated by three hyphens '`---`'), where the first define

a deployment and the second exposes this deployment through the use of a LoadBalancer Service.

- We still have the same issue as before with the use of socket.io and the load balancer, so for now we keep the number of replicas to 1. Giving this instructions means that Kubernetes will create a '`ReplicaSet`' to keep track of the pods in this deployment.
- The Deployment named '`qfstandalone`' has a one template for a pod, with the label '`qfapp`'. As it happens, the selector instructs the ReplicaSet to manage pods matching this same label.
- The pod template says that there should be one container in there, with the name '`qfstandalone`' and based on the image '`mickesv/qfstandalone:version1`'.
- Notice that the '`containers`' is a list, so it is possible to add other containers into the same pod.
- The Service is created to act as a LoadBalancer against the ReplicaSet Deployment. Internally, it will access port 3000, but outside of the cluster (or from other pods inside the cluster), this will be accessed via port 30001.

Cave! Kubernetes will fetch the images from Docker hub, so you either need to push your own qfstandalone image thither or you can trust me (!) and use the '`mickesv`' version.

Tasks

0. Open up a terminal and enter your `QuoteFinder` directory. Create a new sub-directory, e.g. `QuoteFinder/Kubernetes` , and cd into it.
1. Create the file `qfstandalone-v1-1.yaml` and copy the configuration from above into it.
2. Apply the configuration: `kubectl apply -f qfstandalone-v1-1.yaml`
3. Check what was created: `kubectl get all`

1. It is time to test if the web page is available. In order to access the web page, we first need to find the external ip of the qfapp-service (Remember that the service is required to bridge from the outside world to inside the Kubernetes cluster). Get the list of services through `kubectl get services` , and you will notice that each service has both a '`CLUSTER-IP`' and an '`EXTERNAL-IP`'. Right now, the service only has a Cluster-ip, which is internal to Kubernetes. Other containers inside the same Kubernetes cluster may use this to access the qfapp-service, but we cannot use this address from the outside. For this, we need the external IP, and this may still be listed as '`<pending>`'.

On Windows or Mac with Docker Desktop you may have better luck. Since the LoadBalancer service was created with just a `nodePort` specified, Kubernetes will open this port on the node (Right now, you only have one node; your host computer). Thus, the QuoteFinder app may be available as `http://localhost:30001/` .

On Linux with Minikube, the service is exposed with `minikube service qfapp-service` . As a courtesy to you, minikube will open this web page for you.

5. Previously we were able to see the containers print out all sorts of debug info as it was running. Let's try recreating that. First, list all running pods (just the names): `kubectl get pods` . Notice that the name you specified has been appended with some sort of hash key. All of this is the name of the pod.

The command to access the logs in a pod is `kubectl logs -f <podname>`, where the `-f` flag means that we want to *follow* the logs and continue to print anything that is added to them. With a bit of Unix elbow-grease, the following incantation should both find the name of the pod and attach itself to the log: `kubectl get pods -o name | grep qfstandalone | xargs kubectl logs -f` .

Breaking down the command, we see the following:

- The three parts are combined with Unix pipes (the `|` symbol). The output (standard output, to be precise) of the command to the left of the pipe will be the input (standard input) for the command to the right of the pipe. This is how simple Unix commands are daisy-chained to each other to produce ever more complex tasks.
- `kubectl get pods -o name` prints the names (only the names) of all running pods.
- `grep qfstandalone` filters the output to only those containing the string "qfstandalone"
- `xargs` converts whatever it receives on standard input into arguments at the end of whatever command is given to it.
- `kubectl logs -f` is the start of this command, to which xargs add the names of all pods containing the string "qfstandalone" in their names.

The end result is that we are now attached to the logs and can see the same type of output in the console as we were able to do previously. Incidentally, you may notice that users seem to connect from some strange IP address; while this *should* be the address of your host computer, it clearly is not. Take a moment to consider the deployment and see if you can't figure out whose IP address this is.

6. While you are attached to the logs try searching for something on the web page. This will not work since we have not yet connected our database or told the qfapp container where to find it, but the log output is interesting. (you will, for example, get a warning that '`TEXTSTORE_HOST`' is not set).

Summary At this point we have:

1. Started a local Kubernetes Cluster consisting of a single Node.
2. Defined a Kubernetes ReplicaSet Deployment where containers matching the label '`app: qfapp`' is deployed.
3. Re-used the same QuoteFinder v1 (qfstandalone) docker image as before and deployed it on this cluster.
4. Started a Kubernetes LoadBalancer Service to enable access from outside the Kubernetes cluster to the ReplicaSet deployment.

Cleanup Clean up your deployment: `kubectl delete --cascade='foreground' -f qfstandalone-v1-1.yaml` .

13.3 Attach the Database

Attaching a database should in theory be easy, but let's take a moment and do this "right", so that we get a configurable and by the books deployment.

First, of course, the database need to be started. Since we want data to survive restarts, we will do this as a `StatefulSet` . This StatefulSet will launch an image of MongoDB (as before). It will also make two persistent VolumeClaims (as before); one for MongoDB's configuration and one for the data.

These Persistent Volume Claims are then matched to Persistent Volumes by Kubernetes, and the recommendation is that these are specified in a separate file, so that they can be deployed by someone with the right access privileges. App developers make claims, and node administrators provide resources. A Persistent Volume is a resource, and thus need a node administrator to configure.

For now, however, there is no need to specify these Persistent Volumes; at least `minikube` will create them as needed. Other tools may not, and once you deploy to a cloud provider, you most certainly will need to define the Persistent Volumes somewhere and somehow. For now, however, we can leave this.

What we *do* need to take care of, however, is a little snag in how Persistent Volume Claims work. Let's say we decide that we want more replicas of our database container in the StatefulSet. Unless otherwise specified, Kubernetes will map all of these replicas *to the same* volume claim, and MongoDB will most certainly not like this. The first replica will lock the database files, and no other replicas will be allowed to access them.

To avoid this, we will not make concrete claims to a particular persistent volume; we will specify a *template* for the claims, called a `volumeClaimTemplate`. This template specifies the *type* of volume required, but as little else as possible. Especially avoid specifying any storageClassName, or any StorageClasses or PersistentVolumes at all.

Almost finally, in order to access the replicas of the database, we need a Service, and any access (even from within the Kubernetes cluster) will be done with the help of this service.

We still need to configure the QuoteFinder app so that it can find and connect to the database. One Kubernetes way to do this is via a `ConfigMap`. This is a simple key-value map that can be referenced from the container templates.

Enough talking! Time to get busy.

Tasks

0. Copy the previous yaml file to a new file, to keep a record of the different steps: `cp qfstandalone-v1-1.yaml qfstandalone-v1-2.yaml`
1. Add the following to the end of this new copy:

```
# TextStore
# --------------------
# - one container per pod running the image mongodb
# - one headless service to access them.
# - Two persistentVolumeClaims/mounts: textstore-data, and mongo-config
---
apiVersion: v1
kind: Service
metadata:
  name: textstore-service
  labels:
    app: textstore
spec:
  ports:
  - port: 27017
    targetPort: mongodb-port
  selector:
    app: textstore
---
apiVersion: apps/v1
kind: StatefulSet
metadata:
  name: textstore
  labels:
    app: textstore
spec:
  serviceName: mongodb
  replicas: 1
  selector:
    matchLabels:
      app: textstore
```

```
  template:
    metadata:
      labels:
        app: textstore
    spec:
      containers:
      - name: textstore
        image: mongo
        ports:
        - containerPort: 27017
          name: mongodb-port
        volumeMounts:
        - name: textstore-data
          mountPath: /data/db
        - name: mongo-config
          mountPath: /data/configdb
  volumeClaimTemplates:
    - metadata:
        name: textstore-data
      spec:
        accessModes: ["ReadWriteOnce"]
        resources:
          requests:
            storage: 10Mi
    - metadata:
        name: mongo-config
      spec:
        accessModes: ["ReadWriteOnce"]
        resources:
          requests:
            storage: 5Mi
---
```

Especially interesting here is the `volumeClaimTemplates` . As you notice, the claims are made in terms of how we are going to access them ("ReadWriteOnce"), and their desired size. The `metadata.name` match the `volumeMounts` in the container spec above.

2. Also append the following:

```
# ConfigMap to store TEXTSTORE_HOST variable in
# --------------------
---
apiVersion: v1
kind: ConfigMap
metadata:
  name: qfapp-config
  labels:
    data: config
data:
  textstore_host: "NOTSET"
---
```

Caution, `textstore_host` MUST be "NOTSET" (case sensitive). This will be replaced with the right IP during startup.

3. Some edits are required for the qfapp container template to make use of this ConfigMap to set an environment variable (using the `env` specification):

```
spec:
  containers:
  - name: qfstandalone
    image: mickesv/qfstandalone:version1
    ports:
    - containerPort: 3000
    env:
    - name: TEXTSTORE_HOST
      valueFrom:
        configMapKeyRef:
          name: qfapp-config
          key: textstore_host
```

4. Time to deploy all of this. This needs to be done in stages. First, the ConfigMap and the textstore database are created. Then, the Cluster-IP address for the database is extracted so that the ConfigMap can be updated accordingly. Finally, the qfapp deployment can be launched:

```
# Apply only the parts matching the label for the configmap and the textStore
kubectl apply -f qfstandalone-v1-2.yaml -l data=config
kubectl apply -f qfstandalone-v1-2.yaml -l app=textstore

# Extract the clusterIP
export textstoreip=$( kubectl get services/textstore-service --template='{{.spec.cluste

# Retrieve the ConfigMap, replace "NOTSET" with the clusterIP, and re-apply.
kubectl get configmap/qfapp-config -o yaml | sed -r "s/NOTSET/$textstoreip/" | kubectl

# Finally, start the qfapp
kubectl apply -f qfstandalone-v1-2.yaml -l app=qfapp
```

. . . If I were you I would seriously considering making a script for this, since it is only going to get worse.

5. Attach a log to the running instance of qfstandalone: `kubectl get pods -o name | grep qfstandalone | head -1 | xargs kubectl logs -f`
6. Using a separate terminal, open a web browser to your service: `minikube service qfapp-service` (or equivalent command for your version of Kubernetes) and search for something. Note in the log that the connection to the database is established and the search is executed as it should. Do try to add some texts and re-do the search if you like.

Summary We have now:

1. Added a database container and a service to access it.
2. Added a ConfigMap where we can keep the internal IP-address to this service.
3. Updated the qfstandalone deployment to make use of this ConfigMap.
4. Deployed in three stages; start ConfigMap and textstore, extract the Cluster-IP for the text-Store and update the ConfigMap, and started the qfstandalone app and its corresponding service.

Cleanup As before, clean up your deployment: `kubectl delete --cascade='foreground' -f qfstandalone-v1-2.yaml` .

14 Quiz: Deployment with Kubernetes

TODO Please read the following articles and resources:

- Andersson, J., & Norrman, F. (2020). Container Orchestration: the Migration Path to Kubernetes. (Bachelor Thesis)
- Kubernetes Failure Stories (`https://k8s.af/`) (last checked 2023-03-30). This is not to dishearten you from using Kubernetes, but there is a lot to learn by looking at mistakes others have made. Also, in the midst of all this failure there are some examples of really good deployment practices in there too.
- J. Tigani Big Data is Dead (`https://motherduck.com/blog/big-data-is-dead/`) (last checked 2023-03-30). An interesting discussion piece about whether we in fact have moved beyond big data already. Not strictly related to Kubernetes and microservice development, but still a relevant read.
- D. Glasser, MongoDB queries don't always return all matching documents. (`https://blog.meteor.com/mongodb-queries-dont-always-return-all-matching-documents-654b6594a827#.59pxgubtq`) (last checked 2023-03-30) This is just one example of an article that explains challenges with nosql databases and their "eventually consistent" philosophy.

TODO Summarise each article/resource (no more than 1/2 page each) according to the following:

- Authors (if available/applicable) and Title of the article
- Briefly, what is the article about?
- What have they measured?
- What are their main findings?
- What can you learn from this article?

TODO Please answer the following questions:

1. What types of Kubernetes '`Services`' are there?
2. It is recommended to start Services before the Containers they refer to. Why?
3. What is a '`NodePort`'? How does it differ from a '`LoadBalancer`'?
4. What is a '`StatefulSet`'? When is this used?
5. What are the differences between a '`Pod`', a '`Deployment`', a '`ReplicaSet`', and a '`StatefulSet`'?
6. When do you use a '`ConfigMap`'? How does it differ from a '`Secret`'?
7. What are the different ways in which you can use a '`ConfigMap`' to configure a Container? When would you use each of them?
8. Do you have a better solution than the multi-stage startup in order to get the IP address of the textstore-service into the ConfigMap before starting qfstandalone?

There is a Quiz on Canvas where you can submit your article summaries and answers to the questions.

NOTICE: *This quiz does not contribute to the grade in the course. We do, however, require of you to submit the quiz on time.* The purpose of this quiz is to serve as a learning aid allowing you to think about these concepts, and for us to keep track of your progress in the course. If you are unable to maintain the study pace required to submit this quiz on time, we want to be made aware of this so that you are able to re-plan your commitment for the remainder of the course.

15 Scaling the Database

15.1 Database Scaling

```
 ---------------------------
< Can we scale the database? >
 ---------------------------
        \   ^__^
         \  (oo)\_______
            (__)\       )\/\
                ||----w |
                ||     ||
```

Yes. Yes, we can. The first and easiest way to scale is by scaling the node it runs on, i.e. *vertical scaling*. This implies increasing the number of CPUs, the amount of RAM, and/or increase the storage capacity.

The second way to scale is by *replication*. This creates duplicate copies of the entire database, each replica requiring as much resources as a single instance of it. Typically, this is combined with a load balancer to even out the load between the replicas. This is typically good if the application *reads* data more often than it *writes*. Writing data will be slower since the edit will need to propagate across all replicas before it becomes available, but reading can be done by any single replica. In contrast to the classic relational databases, most NoSQL databases (such as MongoDB), only promise *eventual consistency*, and this is put to good use for read-intensive applications with database replication.

The *third* way of scaling is to split the database across multiple nodes. This, in turn, can be done in some different ways. First, one may divvy up the tables across the available nodes. For simple queries it may be sufficient to just hit a single node, but for more complex joins it may be necessary to perform several queries to get data from tables distributed to other nodes as well. Simplifying a little (ok; a lot) one may see this as *splitting the database across columns*.

When, instead, the database is split across rows so that some *rows* are available on one node and other rows are available on another node, this is called *Sharding*. The simplest way to do this is to specify ranges for one key and assign a shard ID for each range. Stepping up, one may have an arbitrarily complex algorithm to compute the shard ID for each database record.

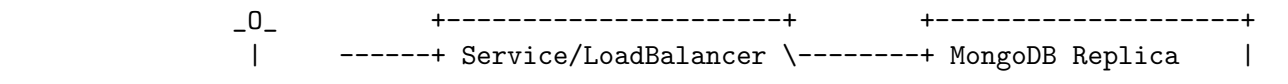
Abandoning all hopes of being able to apply Boyce-Codd Normal Form, it is of course desirable to keep related data together on the same node. Rather than splitting data across tables and use joins when querying, each record can be designed to contain all the necessary data. These records can then be distributed based on some geographical key, for example to comply with laws governing where citizens' data must be located.

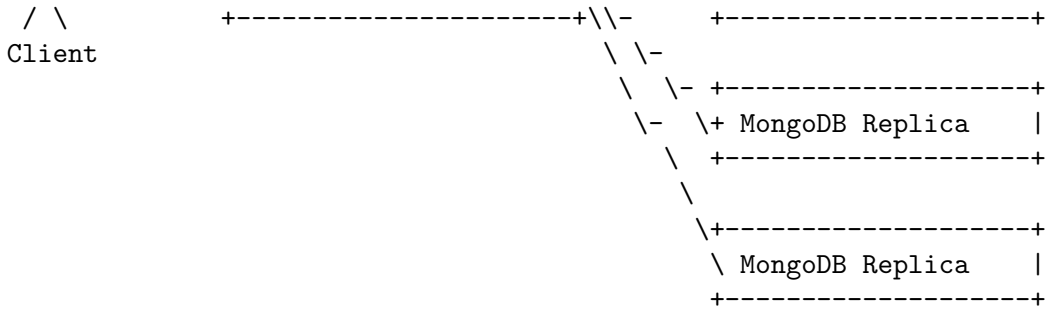
Of course, one can split across columns as well through sharding. One may even use different sharding architectures for each table. Madness only ever lies a single design decision away.

15.2 MongoDB ReplicaSet

For the sake of selling access to various tools, everyone agrees that Database Sharding is complex and painful. So in this course we will focus on the next best step, which is to set up a *replicated* database. We will get some help from Kubernetes, but since each NoSQL database implements this in their very own creative way this is also going to involve a certain amount of tool specific (MongoDB) incantation.

Conceptually, replication ought to work as illustrated below:





In practice, however, it doesn't. MongoDB does not know that there is such a thing as external Load-Balancers (and in any case it can not rely on that the user knows). The MongoDB replica do also not want to assume this role, and if they did they would still need to decide which IP should be broadcast to the outside. Instead, the design solution that MongoDB has opted for is to push load balancing onto the database clients.

By now you know that I have plenty of opinions about software and design decisions so I will leave it as an exercise to the reader to define, list, and sort a suitable array of invectives.

15.3 Setting up a MongoDB ReplicaSet in Kubernetes

The following items from the previous setup are generic enough that they do not require any modification at this stage:

The qfapp-service
: this service only enables access from the outside of the Kubernetes cluster, so there is no need to change it.

The textstore-service
: this too will remain unchanged. To be honest, I'm not sure whether it is actually used, but it may need to be there anyway.

The volumeClaimTemplates for the textstore containers
: These are the same as before and will remain unchanged.

1. TextStore StatefulSet

   The '`textstore StatefulSet`', does require some changes:

   - For cosmetic reasons, we replace the '`serviceName`' to "textstore-service". This becomes part of the hostname (e.g. `textstore-0.textstore-service.default.svc.cluster.local`)
   - The number of '`replicas`' starts out with the value '`3`'.
   - The startup '`command`' needs to be edited to start MongoDB in ReplicaSet mode: '`["mongod", "--bind_ip", "0.0.0.0", "--replSet", "MainRepSet"]`' . Breaking this down:
     - listen to all network interfaces '`--bind_ip 0.0.0.0`'
     - use the specified name for the ReplicaSet '`--replSet MainRepSet`'
   - A *Readiness Probe* is defined. This will be used by Kubernetes to check whether the startup of a replica is ready and it can be added to the list of running replicas. Running a "ping" command inside the database is usually a good start for this.

   In summary, the following yaml defines the textstore StatefulSet:

```
---
apiVersion: apps/v1
kind: StatefulSet
metadata:
```

```
  name: textstore
  labels:
    app: textstore
spec:
  serviceName: textstore-service
  replicas: 3
  selector:
    matchLabels:
      app: textstore
  template:
    metadata:
      labels:
        app: textstore
    spec:
      containers:
      - name: textstore
        image: mongo
        command: ["mongod", "--bind_ip", "0.0.0.0", "--replSet", "MainRepSet"]
        readinessProbe:
          exec:
            command: ["mongosh", "--eval", "db.adminCommand({ping: 1})"]
        ports:
        - containerPort: 27017
          name: mongodb-port
        volumeMounts:
        - name: textstore-data
          mountPath: /data/db
        - name: mongo-config
          mountPath: /data/configdb
  volumeClaimTemplates:
    - metadata:
        name: textstore-data
      spec:
        accessModes: ["ReadWriteOnce"]
        resources:
          requests:
            storage: 10Mi
    - metadata:
        name: mongo-config
      spec:
        accessModes: ["ReadWriteOnce"]
        resources:
          requests:
            storage: 5Mi
---
```

2. Qfapp ConfigMap

   Accessing the database, will require a new access string:

```
---
apiVersion: v1
kind: ConfigMap
metadata:
```

```
    name: qfapp-config
    labels:
      data: config
  data:
    textstore_host: "NOTSET"
    textstore_replicaset: "mongodb://textstore-0.textstore-service.default.svc.cluste
  ---
```

I am not a fan of that the URLs to each member in the cluster has to be hard-coded like this ('`textstore_replicaset`'), since this limits the ability to scale up or down the number of replicas. Ideally, it should be as easy to change the '`spec.replicas`' for the StatefulSet and let Kubernetes re-apply. In practice, this is now made much more difficult.

3. QFStandalone Deployment

   As a start, the only edit to the qfstandalone deployment we make is to include the newly defined element from the ConfigMap. Specifically, the environment variable '`TEXTSTORE_HOST`' need to get its value from '`textstore_replicaset`' in the configmap.

```
---
apiVersion: apps/v1
kind: Deployment
metadata:
  name: qfstandalone
  labels:
    app: qfapp
spec:
  replicas: 1
  selector:
    matchLabels:
      app: qfapp
  template:
    metadata:
      labels:
        app: qfapp
    spec:
      containers:
      - name: qfstandalone
        image: mickesv/qfstandalone:version1
        ports:
        - containerPort: 3000
        env:
        - name: TEXTSTORE_HOST
          valueFrom:
            configMapKeyRef:
              name: qfapp-config
              key: textstore_replicaset
---
```

   Do note that this will not be quite sufficient to create a workable solution. Can you already now figure out why this will not be sufficient?

4. Tasks

   0. Copy the previous yaml file to a new file, to keep a record of the different steps: `cp qfstandalone-v1-2.yaml qfstandalone-v1-3.yaml`
   1. Locate the '`textstore StatefulSet`' and replace it with the version above.

2. Locate the '`QFApp ConfigMap`' and replace it with the version above.
3. Locate the '`QFStandalone Deployment`' and replace it with the version above.
4. Start everything: `kubectl apply -f qfstandalone-v1-3.yaml`
5. Wait for the replicas to come online: `kubectl rollout status statefulset textstore --timeout=30s` This may take longer than 30 seconds, so you may need to run the command a couple of times. If it *does* take longer, it is an excellent opportunity to open up Kubernetes' dashboard and see what is going on: '`minikube dashboard &`'
6. Once all three replicas are running (but not before!), it is time to initiate the replicaset in MongoDB. Run the following: `kubectl exec -it textstore-0 -- mongosh --quiet --eval "rs.initiate()"` This will log in to *one* of the members in the replicaset and initiate it. Sometimes, MongoDB has already figured this out, and you will receive a message "already initialized". This is ok.
7. Double check the status: `kubectl exec -it textstore-0 -- mongosh --quiet --eval "rs.status()"` One particular thing to look out for is to make sure that one of the servers has the `stateStr: 'PRIMARY'` . You may have to wait a while and check again for this.
8. Connect to the logs in qfstandalone (as before): `kubectl get pods -o name | grep qfstandalone | head -1 | xargs kubectl logs -f`
9. Connect to the web interface (e.g. with `minikube service qfapp-service`) and search for something. Keep an eye on the logs to see why it fails.

It appears that the current implementation of the database connection in '`simpleTextManager.js`' is trying to add too much to the environment variable. It would be nice if we were able to enter the container and edit the running application as we did before, but for reasons this is not easily done (see [kube-edit], page 58,). With a bit of elbow-grease we can at least get things to work decently (don't even bother trying to do some fancy development in this way).

10. Find the name of the qfstandalone pod: `kubectl get pods -o name | grep qfstandalone`
11. Enter into this pod: `kubectl exec -it <podname> -- sh` (where "podname" is the name of the pod as found in the previous step. Feel free to poke around at your own leisure here and see what you can find and what works. Note that in order to keep the container image small we used a very limited linux distribution as base, so there isn't much installed at all.
12. Edit the file `/app/src/simpleTextManager.js` (I'm afraid the only editor you've got is '`vi`') . Replace line 20 with `let connection=process.env.TEXTSTORE_HOST;` and save. You should see the app restarting in the log output you attached to earlier, and if you try searching for something you should no longer get any errors.

Summary We have now:

1. Configured the simplest and unsafest possible MongoDB replicaset without any form of authentication whatsoever.
2. Created a single-phase startup where everything can be deployed with a single '`kubectl apply`' command.
3. Waited for parts of a deployment to come online in an orderly fashion with `kubectl rollout status` and a ReadinessProbe.
4. Ugly-hacked our way into a container to edit the running application code.

Cleanup As before, clean up your deployment: `kubectl delete --cascade='foreground' -f qfstandalone-v1-3.yaml` .

16 Edit a Container in a Pod in a Node in a Cluster

```
 ---------------------------------------
( So why were we not able to use a Bind )
( Mount to access the innards of the    )
( container as before?                  )
 ---------------------------------------
        o   ^__^
         o  (oo)\_______
            (__)\       )\/\
                ||----w |
                ||     ||
```

Technically, we *are* able to do this. The problem is that Kubernetes adds an extra layer. Remember that Containers run inside Pods. So while it is relatively easy and similar to what we have done before to set up a bind mount, this will only allow files to be shared between the insides of a container and the pod it is running on. The next necessary step is to somehow create a bridge from the pod's file system to the host's file system.

The first bit is similar to what we have done before: setup the container template to mount `/app/src`.

```
containers:
- name: qfstandalone
  image: mickesv/qfstandalone:version1
  volumeMounts:
  - mountPath: /app/src
    name: source-dir
```

This is coupled with a matching `volumes` statement for the pod. In the most basic, just create an empty directory inside the pod to match this volume:

```
volumes:
- name: source-dir
  emptyDir: {}
```

However, since the goal was to be able to reach the directory from the outside, we can go for the moonshot with a HostPath instead:

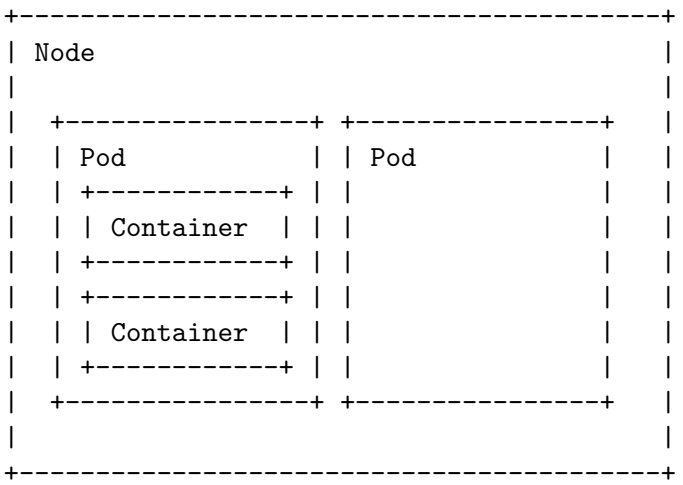
```
volumes:
- name: source-dir
  hostPath:
    path: /app/src
    type: Directory
```

(The path can of course be anywhere here, it does not have to be `/app/src` since that only really makes sense inside the container. However, it is good to keep it short for the next step. In fact, it would not be a bad idea to keep it under `/tmp`.)

Unfortunately, this *still* does not mean that we are able to access the files. The reason for this is a Kubernetes concept which we have not discussed yet, namely that of a *Node*. A Kubernetes node can be seen as the unit that would be run on one piece of (virtual) hardware. On this node pods, volumes, etc. are deployed. `minikube` is a single-node implementation of Kubernetes so there is not much more configuring to do, but if you deploy to a cloud provider you would get to decide how many nodes you want to run your application across, as well as how many volumes, pods, services, etc. that you desire.

You can access the inside of the single node with `minikube ssh` ; please take a moment to convince yourself that the minikube node is essentially a complete virtual machine.

The bottom line is that while we are able to access the inside of the running container with a hostPath volume, *we are only able to do so from within the Kubernetes node*. Not quite what we were after, to put it mildly. It is actually worse than that, since when pods get deleted or moved over to another node, whatever was written to the hostPath directory gets left behind (hence why it would make sense to keep the mount point in `/tmp`).



Despite all warnings about why this is a bad idea, if we still want to access the container's inside, there are two options available to us. One is to define a proper volume with a PersistentVolumeClaim for the container. Of course, there is a non-trivial amount of work to set this up in Kubernetes to ensure that an already existing directory on the host computer is made available as a PersistentVolume. A middle ground is perhaps to export the directory as an NFS share (or any other type of network filesystem if you are concerned with the security issues of NFS), and mount this as the container volume.

The other option is to go the other direction and mount a local directory onto the Kubernetes node, and then use a hostPath setup to propagate this mount into the container. `minikube mount` is able to do this, but due to bugs this is not working at the time of writing.

Summary Accessing the running code of a container in Kubernetes is difficult and there are plenty of reasons to advice against it. The recommendation is to do the development using e.g. Docker Compose and only deploy to Kubernetes as a last step when everything is tested and ship-shape. This includes subsequent debugging: There should be nothing in the containers that wasn't there already when the image was created, and so all debugging can and should be done locally outside of the Kubernetes deployment.

17 Additional Concepts

A few loose ends need to be tied up to complete this introduction to Kubernetes. First, we are finally able to solve the previous problems that we had with scaling the frontend. If you remember, the use of `socket.io` meant that it was difficult to use a simple load-balancer to scale the frontend since each client-server connection relied on a socket to be kept alive for the duration. Well, one of the flags that can be given to a load-balancer service is `spec.sessionAffinity: ClientIP` , which means that Kubernetes will stick all sessions originating from the same client IP to the same deployment instance. This is all that is required in order to be able to scale the number of replicas of the qfstandalone deployment.

One simple way to scale is to edit the `spec.replicas` in the yaml file and simply re-apply it. Go ahead; start the dashboard `minikube dashboard &` , edit the yaml file (add the `sessionAffinity` flag above while you're at it), and re-run `kubectl apply -f qfstandalone-v-1-3.yaml` . You should see the number of deployed pods change as a consequence.

You can of course also scale directly via the command line: `kubectl scale --replicas=3 deployment/qfstandalone` .

Keeping an eye on a running deployment is done with the dashboard – you can often find clues in here about what is currently going on – especially if some deployment is not rolling out as it should. If you want to programmatically keep track of the rollout status, there is the `kubectl rollout status` command. Especially when you have configured Readiness Probes (as we did before) or Liveness Probes, this is a useful way to keep tabs on how the scaling operation is going. Readiness Probes should be defined to ensure that new pods are well and truly booted and ready to start working *before* Kubernetes adds them to the set of running deployments. If no Readiness Probes are defined, the pod may be expected to start serving requests before it is actually finished booting.

Liveness Probes are another useful addition, since they allow Kubernetes to automatically keep track of the health of the running deployment, and you may define rules about what should happen when a pod continues to report that it is not alive (or, to be precise, continuously fails to report that it is alive).

The `kubectl rollout` command can also be used to manage the deployment in some more detail. Most notably, it can be used to `pause` and `resume` parts of the deployment.

When we set up the MongoDB ReplicaSet we dodged the bullet of authentication. Technically, there are a number of steps involved in this that you really ought to try your hand at just because. First, the internal cluster should be told to use a specific key to communicate inside the cluster. Second, once the replicaset is up and running (or before? This used to work in one way and in a recent update it no longer works the same way), you should setup users; One admin user to manage the databases and the database cluster, and at least one regular user that clients can use e.g. when querying the database.

The shared key should of course not be bandied around like so much trash: If this key gets out then your database can become compromised by someone adding their own MongoDB instance to your ReplicaSet. As soon as the key is created, we should thus create a Kubernetes *Secret* out of it, and then access this secret from within the containers. This is similar to how you can use data from ConfigMaps in your containers.

To summarise:

```
# Generate and store the key as a Kubernetes Secret
openssl rand -base64 200 > ./replica-set-key.txt
kubectl create secret generic mongodb-bootstrap-data --from-file=mongodb-keyfile=./repl

# Add some users to the database
kubectl exec -i textstore-0 -- mongosh --quiet --eval 'db.getSiblingDB("admin").createU
```

```
kubectl exec -i textstore-0 -- mongosh --quiet --eval 'db.getSiblingDB("admin").auth("r
```

. . . and the configuration of the MongoDB StatefulSet:

```
---
apiVersion: apps/v1
kind: StatefulSet
metadata:
  name: textstore
  labels:
    app: textstore
spec:
  serviceName: textstore-service  # This becomes part of the hostname (e.g. textstore-0
  replicas: 3
  selector:
    matchLabels:
      app: textstore
  template:
    metadata:
      labels:
        app: textstore
        startup: textstore-wait
    spec:
      containers:
      - name: textstore
        image: mongo
        command: ["mongod", "--bind_ip", "0.0.0.0", "--replSet", "MainRepSet", "--auth"
        readinessProbe:
          exec:
            command: ["mongosh", "--eval", "db.adminCommand({ping: 1})"]
        ports:
        - containerPort: 27017
          name: mongodb-port
        volumeMounts:
        - name: textstore-data
          mountPath: /data/db
        - name: mongo-config
          mountPath: /data/configdb
        - name: secrets-volume
          readOnly: true
          mountPath: /etc/secrets-volume
      volumes:
      - name: secrets-volume
        secret:
          secretName: mongodb-bootstrap-data
          defaultMode: 256
  volumeClaimTemplates:
    - metadata:
        name: textstore-data
      spec:
        accessModes: ["ReadWriteOnce"]
        resources:
          requests:
```

```
            storage: 10Mi
    - metadata:
        name: mongo-config
      spec:
        accessModes: ["ReadWriteOnce"]
        resources:
          requests:
            storage: 5Mi
---
```

One final trend to keep an eye on is that of *lambda functions*. They are not always called that, e.g. Microsoft refers to them as Azure Functions or Azure Durable Functions, but the gist is the same. In some ways, this is a natural extension of the microservice architecture, when even a REST API can be considered too much, and all that is needed is a way to replicate and run a function in a distributed way. Typically, the desire is to develop locally and then seamlessly decide whether to run functions within in the same process, perhaps run them on a high-performance computing unit in the same host computer (for example a graphics card), or to scale out and run an arbitrary number of replicas of the function at the cloud provider. This pattern is not always applicable, but for *some* computing tasks it really lends itself to creating a transparent and yet scalable solution.

18 Summary

This is it! We have reached the end of the road for what I wanted to cover in this part of the course. (See Section 3.1 [Learning Outcomes], page 4, for a list of the relevant Learning Outcomes) To begin with, the following concepts have been explored:

- Microservices as a Software Architecture Design Decision
- Principles of Microservice Architectures
- Cloud Application Architecture Patterns
- Microservices and REST APIs

A software architecture consists of the design decisions that underpin it. Once the decision has been taken to build the application as a collection of microservices (and the rationale for this has been captured and documented), there are a number of design decisions that follow, including how separate components in the architecture should collaborate to solve the overall task, and some design principles and business decisions are given as a consequence. To some extent, much of the remaining work is simply to instantiate these already given design decisions in the best possible way, to leverage as much business use as possible out of them.

From these overall design decisions we dive deeper into more specific technologies, i.e.:

- Hypervisors and Lightweight Virtual Machines
- Provisioning using shell scripting
- Provisioning using dedicated provisioning tools
- Orchestration and Deployment of a complete microservice solution

Specifically, we use some specific tools:

- Docker, as a means to define a single microservice
- Vagrant and a Hypervisor, as an alternative to set up a full fledged virtual machine
- Docker Compose, to orchestrate a complete local solution
- Kubernetes, to orchestrate and deploy a solution possibly to a cloud provider

These technologies and tools are typical representatives of contemporary software microservice development. They are by no means alone on the market, there are plenty of other competing technologies, but with this stack and an understanding of the concepts behind the tools you are well equipped to face the future and any tool *du jour* life may throw at you.

While exploring these tools, there is one key principle that have permeated every step along the way: *Infrastructure as Code* . Only by treating the infrastructure, the deployment, in the same way as the rest of an applications code-base, are we able to maintain repeatable orchestrations, where each individual part of a deployment is configuration managed and tested.

In fact, this is important so let me repeat myself:

Infrastructure is Code

Here's a bunny to help you remember:

```
 ------------------------
( Infrastructure is Code )
 ------------------------
      o
(\/) o
(..)
c(")(")
```

The "only" thing remaining for You to do now is to prove that you too have learnt all this. Read on to figure out how.

19 Assignment: Build Something

It is time to put everything you have learnt so far into practice. The goal of this assignment is for you to create a microservice-based application according to the following principles:

1. The application must be deployable using Kubernetes.
2. The application must consist of at least two different types of microservices
3. Each microservice must implement a REST API for access
4. The application must be accessible from "the outside" (e.g. via a web browser)
5. All microservices in the application must be horizontally scalable independently from each other.
6. Any microservice images required to run your application must have been pushed to docker hub so that Kubernetes may find them.
7. The application must make use of some form of database running as a separate microservice.
8. The database must use storage which is persistent across restarts of the deployment infrastructure.
9. The database does not have to be scalable.

The following items shall be delivered:

1. A description of the software you have built and what it does.
2. A software architecture design of your application, describing the role of each component in your system, their responsibilities, and the architecture principles (e.g. cloud architecture patterns) that are used to connect them to a functioning system. This also includes a mapping between software components and the microservices designed and built to implement the components.
3. A discussion of the benefits and challenges with your architecture design. This must include a discussion about security. It must also include a discussion of what you have done or what can be done to mitigate the identified challenges.
4. A link to a configuration management repository (e.g. git) where the source code of the application can be viewed. This must also include the code for your Kubernetes deployment.

Once delivered, *a meeting will be scheduled* where you are expected to run and explain your application and its deployment. Be prepared to answer questions about:

- Your software application idea
- The architecture design decisions in your application
- The business implications of these architecture decisions
- Interaction between different microservices
- Details of your deployment
- Security issues identified and/or mitigated

Your application:

- may implement any software idea; it may also be a copy of an already existing software.
- may be too small or trivial to actually warrant scaling; please acknowledge this and provide instructions for what needs to be pretended for the application to make sense.
- may use any programming language.
- may use a different access interface than a web browser, as long as no additional tools need to be installed on my computer.
- may use any type of database (e.g. NoSQL, SQL, Graph, etc.).
- may be deployed to a cloud provider already.

There is a Quiz on Canvas where you can submit your article summaries and answers to the questions.

NOTICE: This is a *marked* assignment that contributes to the grade on the course. Specifically, the assignment contribute to your grade on the following parts of the course:

- Cloud Provisioning and Deployment
- The Business Case for Cloud Computing

Concept Index

R

S

U

V

X

Program Index

Files and Data Types Index

Functions and Commands Index

D

J

Y