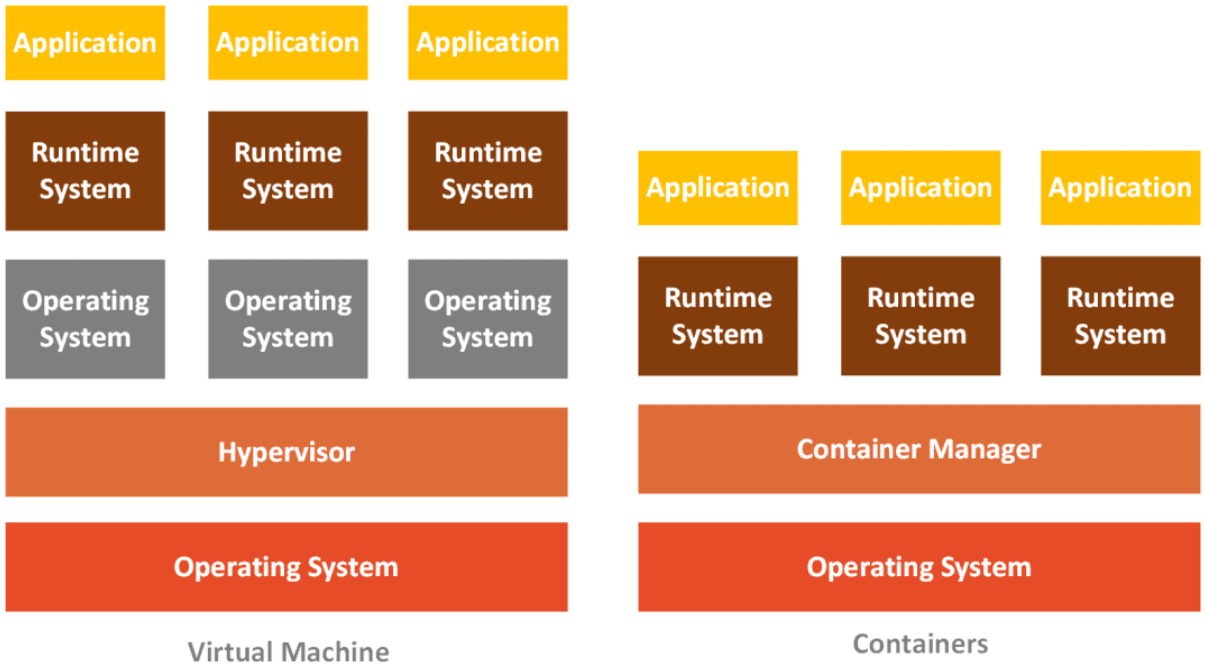
5.4 Docker Basics

In this section, we focus on the **scalability** problem, and we explore model deployment with **Docker**. Docker and **containers** were born out of the need to better utilize servers. Basically, at one moment in history, servers became very powerful in terms of processing power. However, we never create processes there that **require** and utilize that power. Web Applications don't need that. That is when concepts of virtualization and virtual machines were born. These concepts gave servers the power to run more applications on different operating systems at the **same time**. Then, big companies like Amazon saw this as a business opportunity and provided cheap cloud solutions based on it. This is how the **cloud** was created. During this time, applications became bigger in terms of dependencies. It became hard to develop and maintain them because a developer should take care of **all** external libraries, frameworks, and operating systems. Docker and containers solved this problem by providing a means to run any application **regardless** of the operating system.

At this point, one might ask, what is a container? The container packages and unites the application code with all its dependencies. This way, it becomes a **unit** that can run on any computing environment. Docker is a tool that helps us **build and manage** containers. The difference between the container and the virtual machine is in the way that hardware is **utilized**. In general, virtual machines are managed by Hypervisors. However, Hypervisors are only available on processors that support the virtual replication of hardware. This essentially means that virtual machines run software on **real hardware** while providing **isolation** from it. On the other hand, containers require an operating system with basic services and use virtual-memory support for isolation. To sum up, a virtual machine provides an **abstract machine** (along with device drivers required for that abstract machine), while a container provides an **abstract operating system**.



To install Docker, follow instructions provided on this **page**. Docker comes with the UI, which we will not consider in this chapter. We utilize only docker **CLI**, which comes with this installation as well. You know, like real hackers.

There are three important Docker components that you should be aware of: Docker container **Image**, **Dockerfile** and Docker **Engine**. Docker container image is a lightweight file-system that includes everything that the application needs to run. It has the system tools and libraries, runtime and the application code. These images are **turned** into Docker containers once a user runs the *docker build* command, which will be explained in detail in a little while. Once this command is initiated, Docker uses *Dockerfile* to create a Docker container from Docker Image. This container is then run by the Docker engine. There are a lot of pre-cooked Docker Images available at **Docker Hub**. For example, if you have a need for a docker image for *Ubuntu*, you can find it at Docker Hub. Docker image is obtained by initiating command: *docker pullimage_name*. For the purpose of this chapter, we need three images, so let's download them right away:

```
docker pull ubuntu
docker pull tensorflow/tensorflow
docker pull tensorflow/serving
```

Note that every Docker image has multiple **tags**, which can be observed as a specific image **version** or variant.

The *Dockerfile* is the file that defines what the container…well, contains. In this file, a user defines which Docker image should be used, which dependencies should be installed and which application should be run and how. In essence, a Docker image consists of read-only **layers**. Each of these layers is represented with one Dockerfile **instruction**.

Usually, we start a *Dockerfile* with FROM instruction. With it, we define which image is used. Then we use different instructions to **describe** the system and the application we want to run. We may use COPY instruction to copy files from the local environment to the container, or we can use WORKDIR instruction to set the working directory. We can run various commands within the container with CMD instruction and run the application with the RUN command. A full **list of** Docker file instructions and how they can be used can be found **here**.

Let's observe one example of Dockerfile:

```
FROM ubuntu:18.04
COPY . /app
RUN make /app
CMD python /app/app.py
```

In this file, the first instruction – FROM defines that we use ubuntu:18.04 Docker image. COPY instruction adds files to the container's *app* directory. RUN builds the application with *make* and CMD runs command '*python app/app.py*' which in turn runs the application.

Once the *Dockerfile* is ready, we can proceed with **building** and **running** the Docker container. We already mentioned some of the Docker commands that we can use, but let's get into more details. With Docker installation comes the rich *Docker CLI*. Using this CLI we can **build**, **run** and **stop** our container. In this chapter, we explore some of the most important commands. The first on the list is definitely the *docker build* command. When this command is issued, the current working directory becomes a so-called **build context**. All files from this directory are sent to the container. Docker assumes that the *Dockerfile* is located in this directory, but you can define different locations as well. Once this command finishes its job, the user can run the *docker run* command and run the container.

Docker CLI has other useful commands. For example, you can check the **list** of running containers with the command *docker ps*. Also, you can **stop** running a container with the command *docker container stop CONTAINER_ID*. If you need to run some commands within the *bash* of the container, you can use the *docker exec -it /bin/bash* to run the bash. Ok, this is all cool in theory, but let's do something practical. We want to run the Flask application we created in the previous section within the Docker container.

5.5 Deployment with Flask and Docker

First, we regroup our files a little bit into this kind of structure:

```
└──app
    ├──static
    │   ├──css
    │   └──model
    └──templates
```

Because there are several **requirements** we installed with *pipenv*, we need to make sure that these requirements are installed in the container as well. In order to make our lives a little bit easier, we **convert** requirements from pipenv lock file into *.txt* file. This is done with the command:

```
pipenv lock --requirements --keep-outdated > requirements.txt
```

Don't forget to run *pipenv synced -d* before it though. This will create *requirements.txt* which looks something like this:

```
-i https://pypi.org/simple
 absl-py==0.9.0
 astor==0.8.1
 cachetools==4.0.0
 certifi==2019.11.28
 chardet==3.0.4
 click==7.0
 flask==1.1.1
 gast==0.2.2
 google-auth-oauthlib==0.4.1
 google-auth==1.11.0
 google-pasta==0.1.8
 grpcio==1.26.0
 h5py==2.10.0
 idna==2.8
 itsdangerous==1.1.0
 jinja2==2.11.1
 keras-applications==1.0.8
 keras-preprocessing==1.1.0
 markdown==3.1.1
 markupsafe==1.1.1
 numpy==1.18.1
 oauthlib==3.1.0
 opt-einsum==3.1.0
 protobuf==3.11.3
 pyasn1-modules==0.2.8
 pyasn1==0.4.8
 requests-oauthlib==1.3.0
 requests==2.22.0
 rsa==4.0
 six==1.14.0
 tensorboard==2.0.2
 tensorflow-estimator==2.0.1
 tensorflow
 termcolor==1.1.0
 urllib3==1.25.8
 werkzeug==0.16.1
 wheel==0.34.2 ; python_version >= '3'
 wrapt==1.11.2
```