Architecture of Enterprise Applications 22
Virtualization & Container

Haopeng Chen

***RE**liable, **IN**telligent and **S**calable Systems Group (**REINS**)*
Shanghai Jiao Tong University
Shanghai, China
http://reins.se.sjtu.edu.cn/~chenhp
e-mail: chen-hp@sjtu.edu.cn

Contents and Objectives

- Virtualization
- Container
  - Docker
  - Develop Applications with Docker

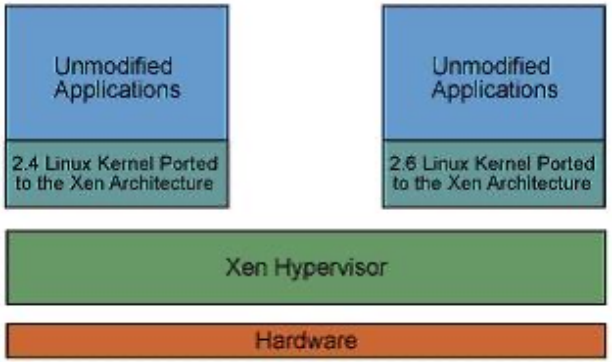
- Objectives
  - 能够根据系统容器化部署的需求，将本地应用迁移到云中容器集群中进行部署

Xen

- Xen Project https://xenproject.org/help/documentation/
- Xen Project virtualization and cloud software include many powerful features that make it an excellent choice for many organizations:
  - Supports multiple guest operating systems: Linux, Windows, NetBSD, FreeBSD
  - Supports multiple Cloud platforms: CloudStack, OpenStack
  - Reliable technology with a solid track record
  - Scalability
  - Security
  - Flexibility
  - Modularity
  - VM Migration
  - Open Source
  - Multi-Vendor Support

Virtualization

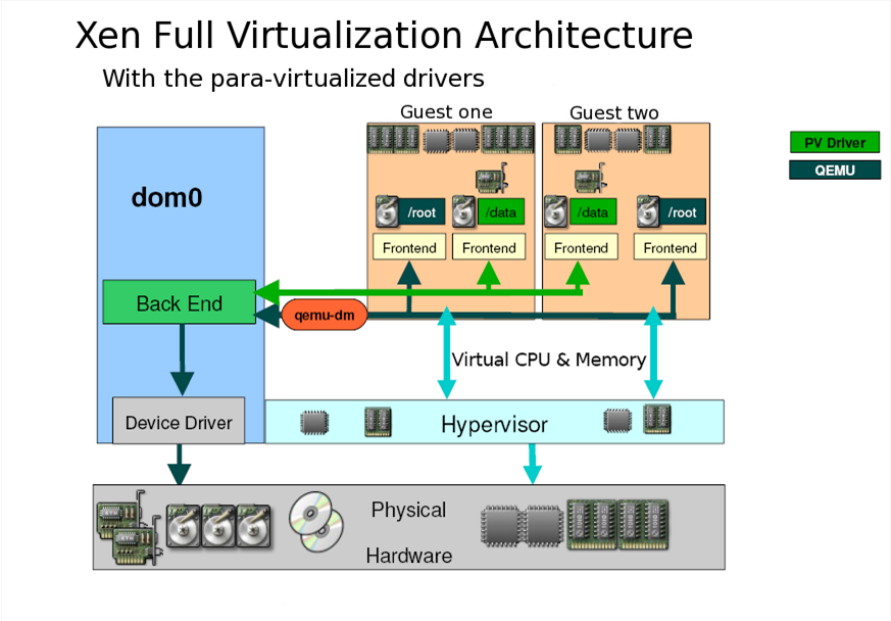
- Paravirtualization vs. Hardware VirtualMachine



- References:
  - https://www.cnblogs.com/sddai/p/5931201.html
  - https://my.oschina.net/davehe/blog/94039

Virtualization

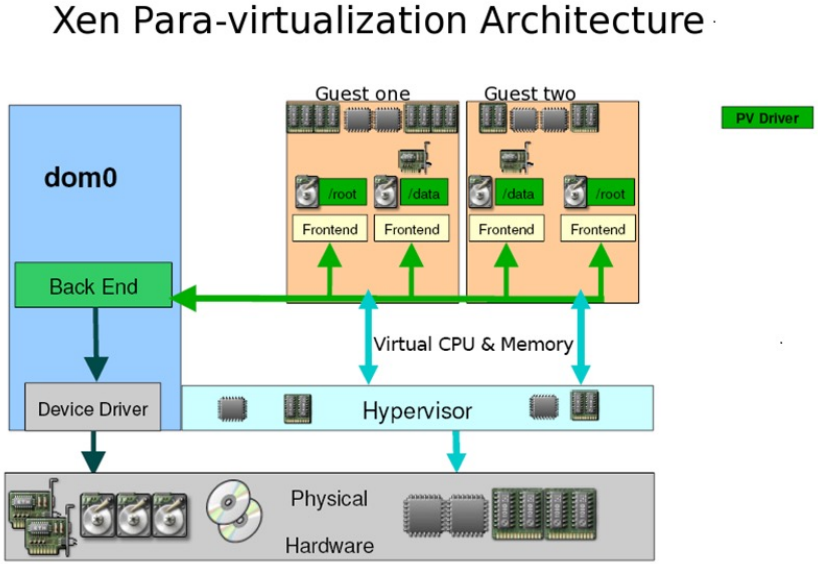
- Paravirtualization vs. Hardware VirtualMachine



- References:
  - https://www.cnblogs.com/sddai/p/5931201.html
  - https://my.oschina.net/davehe/blog/94039

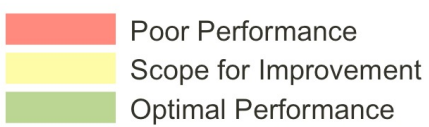
Virtualization

- PVHVM



- References:
  - https://www.cnblogs.com/sddai/p/5931201.html
  - https://my.oschina.net/davehe/blog/94039

Virtualization

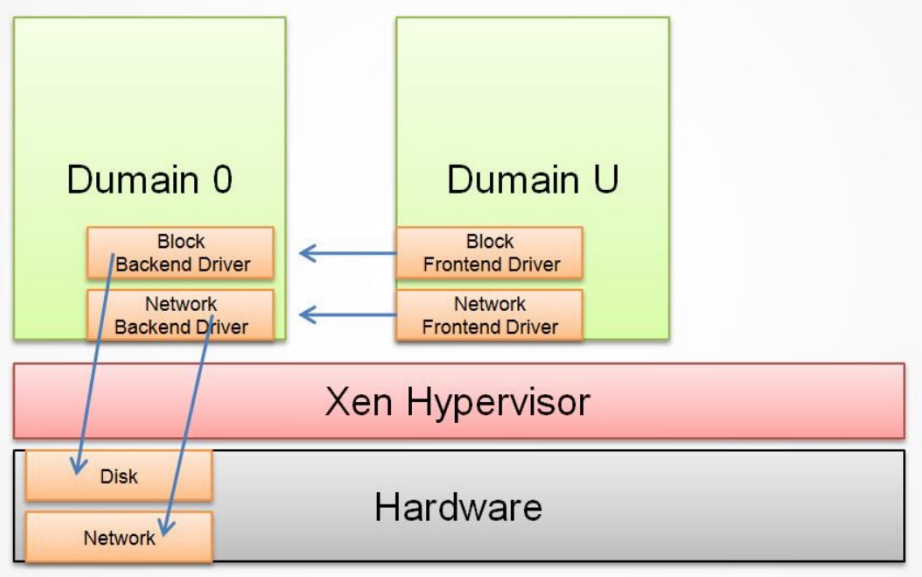


P = Paravirtualized
VS = Software Virtualized (QEMU)
VH = Hardware Virtualized

Shortcut	Mode	With	Disk and Network	Interrupts & Timers	Emulated Motherboard, Legacy Boot	Privileged Instructions, Page Tables
HVM / Fully Virtualized	HVM		VS	VS	VS	VH
HVM + PV drivers	HVM	PV Drivers	P	VS	VS	VH
PVHVM	HVM	PVHVM Drivers	P	P	VS	VH

- References:
  - https://www.cnblogs.com/sddai/p/5931201.html
  - https://my.oschina.net/davehe/blog/94039

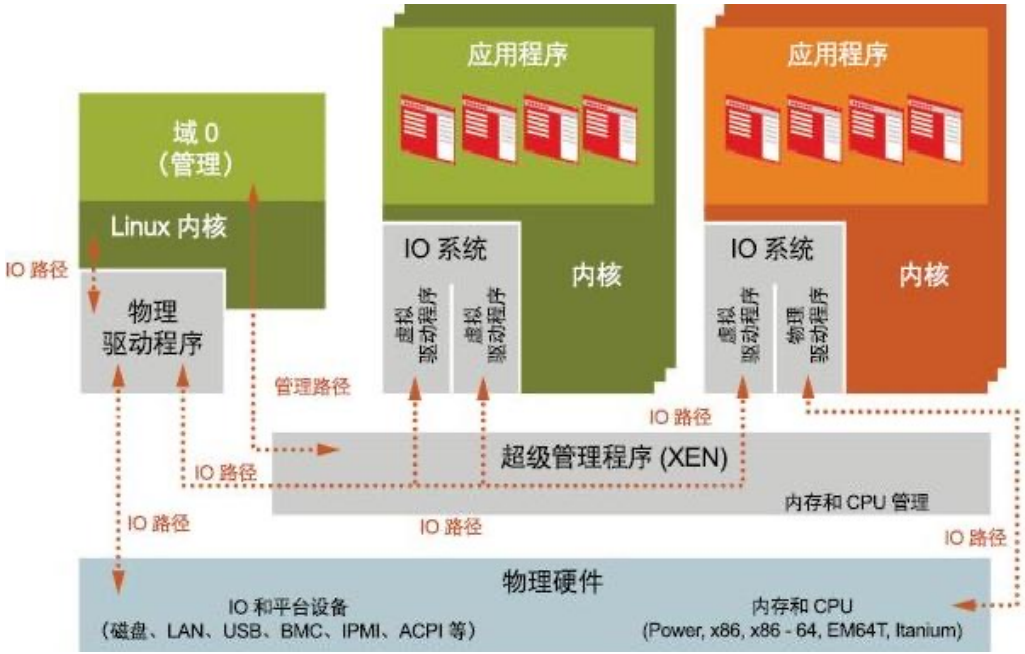
- Xen Components



- References:
  - https://www.cnblogs.com/sddai/p/5931201.html
  - https://my.oschina.net/davehe/blog/94039

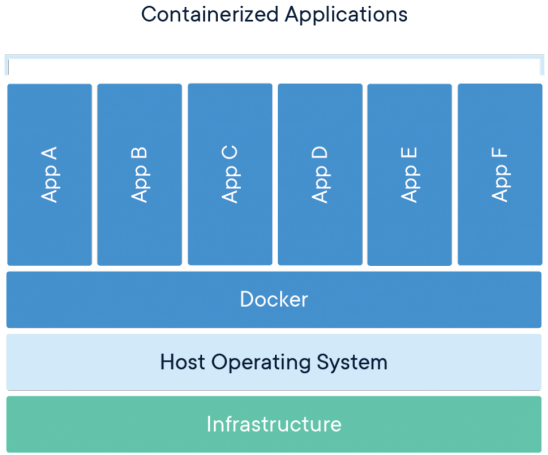
Virtualization

- Xen Architecture



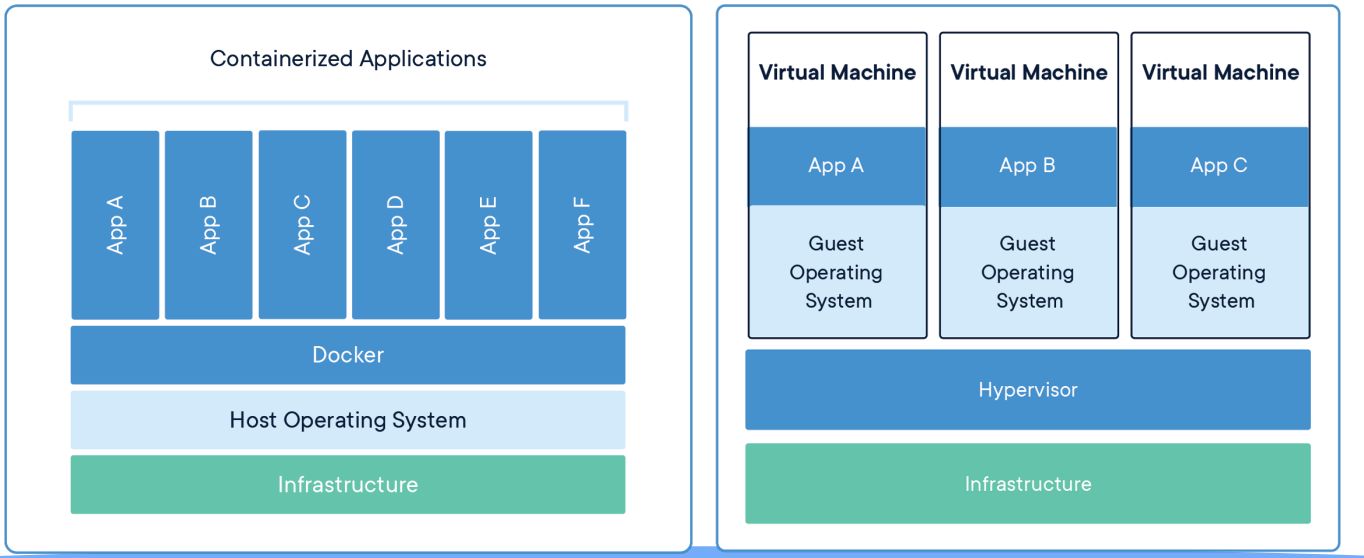
- References:
  - https://www.cnblogs.com/sddai/p/5931201.html
  - https://my.oschina.net/davehe/blog/94039

- **What is a Container?** https://www.docker.com
  - A container is a standard unit of software that packages up code and all its dependencies so the application runs quickly and reliably from one computing environment to another.
  - A Docker container image is a lightweight, standalone, executable package of software that includes everything needed to run an application: code, runtime, system tools, system libraries and settings.
  - Container images become containers at runtime and in the case of Docker containers - images become containers when they run on Docker Engine.
  - Containers isolate software from its environment and ensure that it works uniformly despite differences for instance between development and staging.



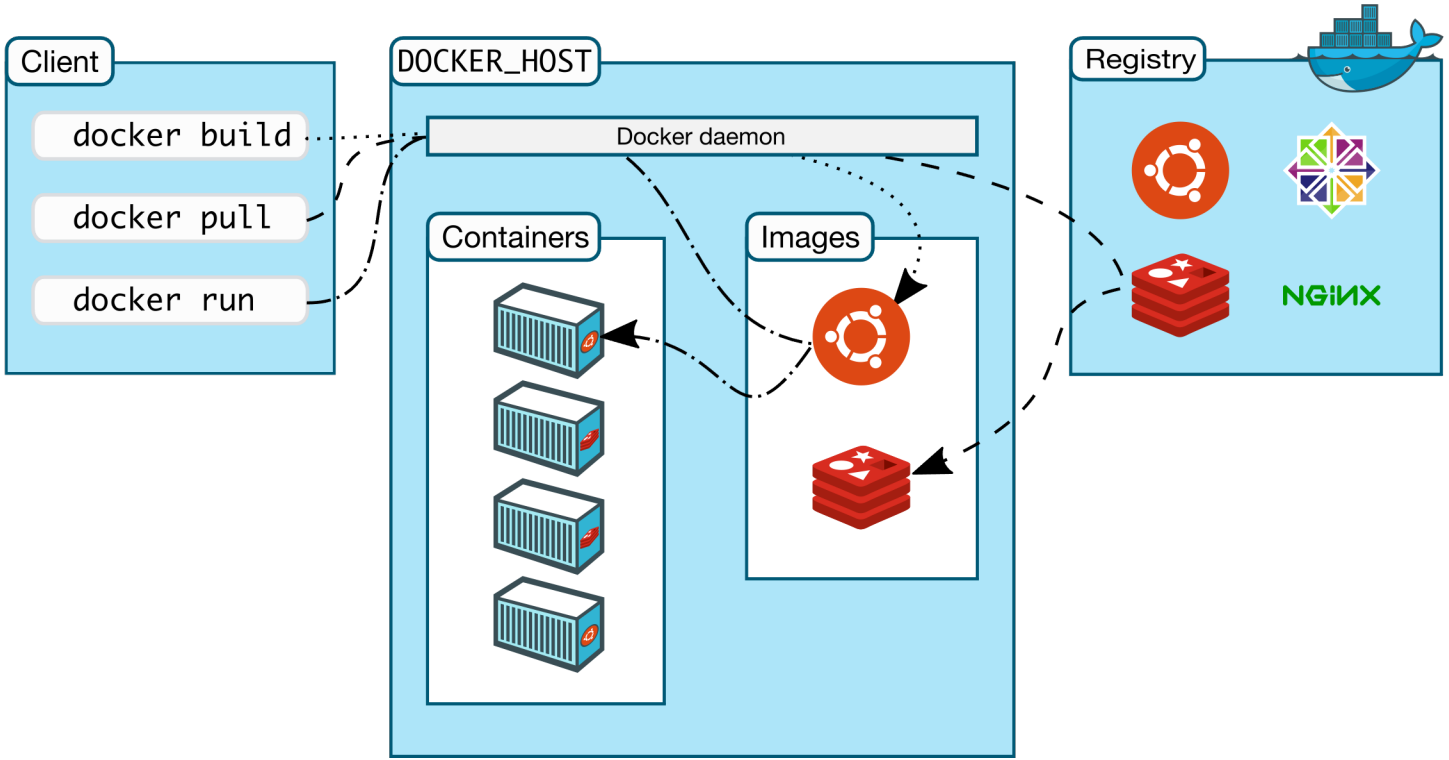
Comparing Containers and Virtual Machines

- Containers and virtual machines have similar resource isolation and allocation benefits,
  - but function differently because containers virtualize the operating system instead of hardware. Containers are more portable and efficient.



Docker Architecture

- Docker uses a client-server architecture

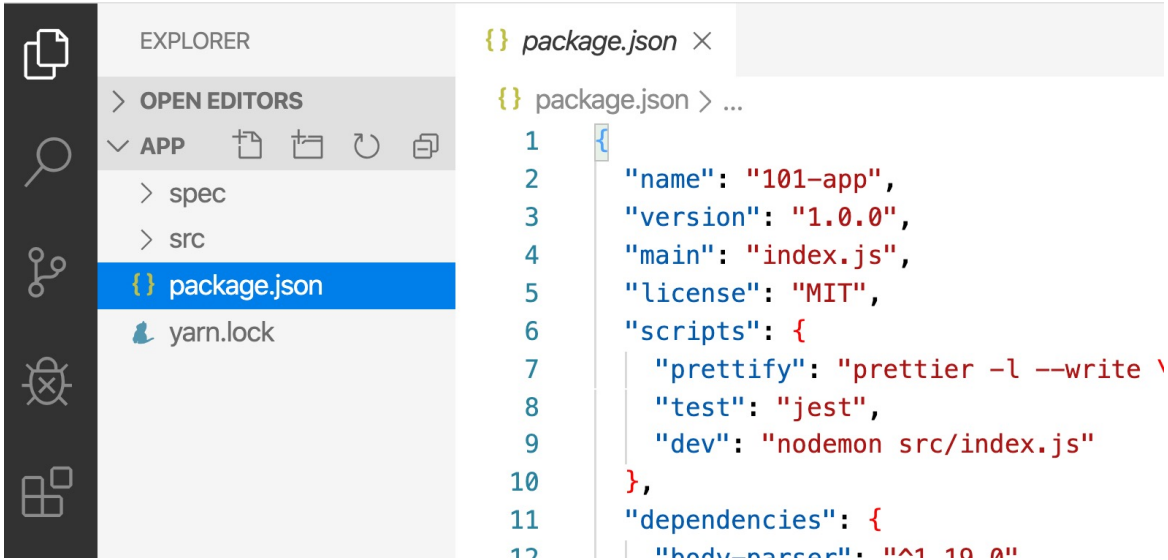


The underlying technology

- Docker is written in the Go programming language and
  - takes advantage of several features of the Linux kernel to deliver its functionality.
- Docker uses a technology called namespaces to provide the isolated workspace called the *container*.
  - When you run a container, Docker creates a set of *namespaces* for that container.
  - These namespaces provide a layer of isolation.
  - Each aspect of a container runs in a separate namespace and its access is limited to that namespace.

Sample application

- *From:* https://docs.docker.com/get-started/
- Get the app
  - https://github.com/docker/getting-started/tree/master/app



Sample application

- Build the app's container image
  - Create a file named Dockerfile in the same folder as the file package.json with the following contents.

```
FROM node:12-alpine
RUN sed -i 's/dl-cdn.alpinelinux.org/mirrors.aliyun.com/g' /etc/apk/repositories
RUN apk add --no-cache python g++ make
WORKDIR /app
COPY . .
RUN yarn install --production
CMD ["node", "src/index.js"]
```

  - Now build the container image using the docker build command.

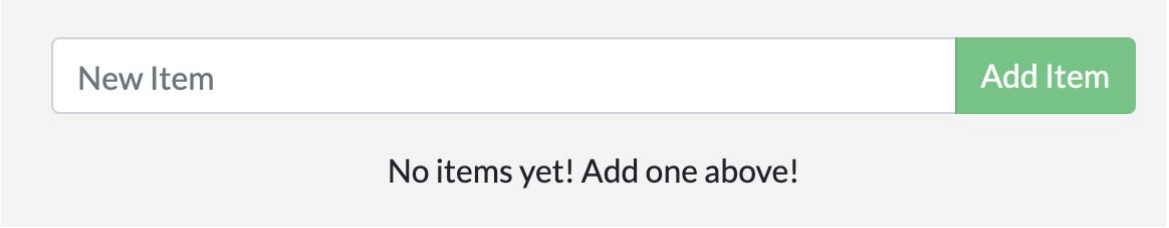
```
docker build -t getting-started .
```

Sample application

- Start an app container
  - Start your container using the docker run command and specify the name of the image we just created:

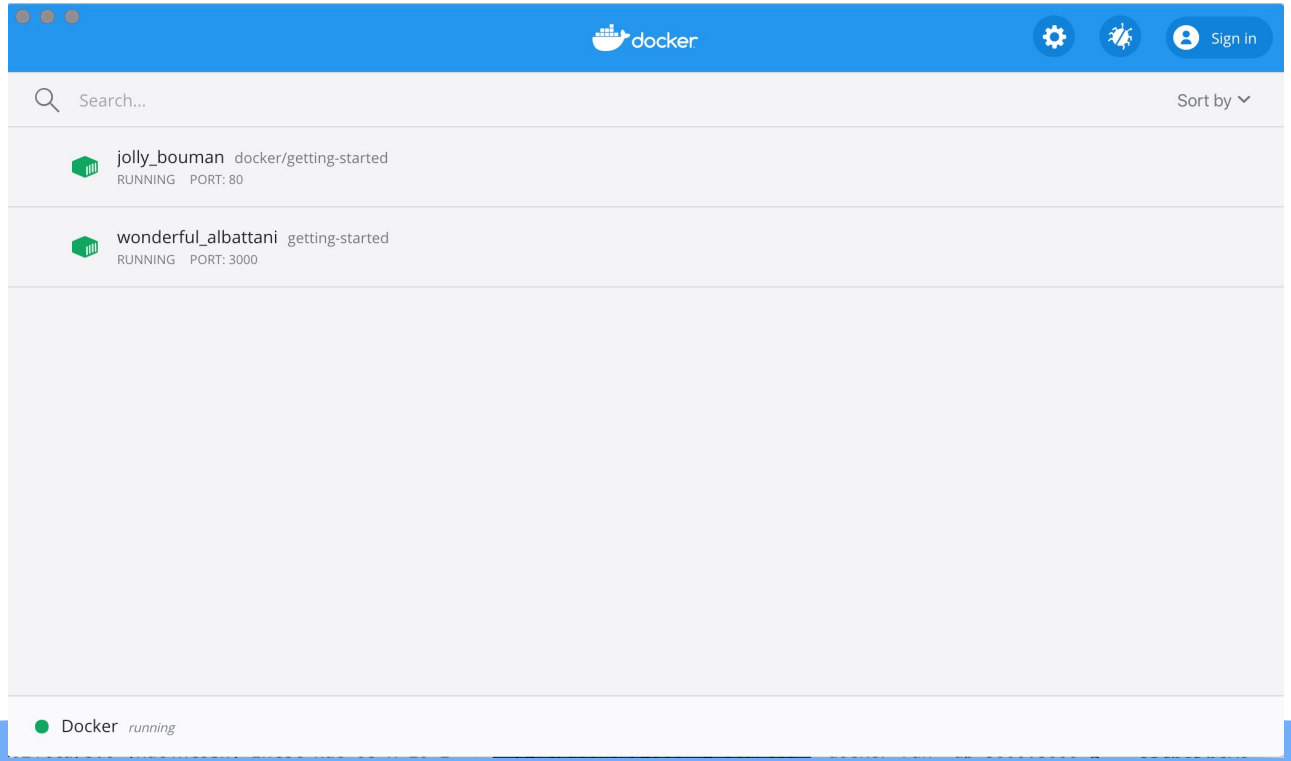
```
docker run -dp 3000:3000 getting-started
```

  - After a few seconds, open your web browser to http://localhost:3000.



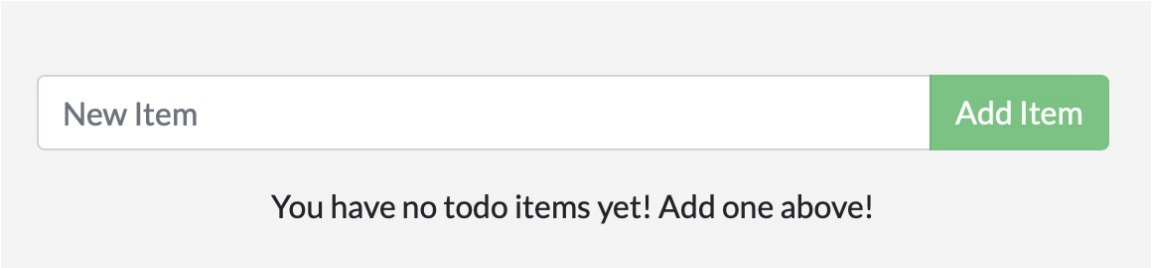
Sample application

- Start an app container
  - Take a quick look at the Docker Dashboard.



Update the application

- Update the source code
  - In the src/static/js/app.js file, update line 56 to use the new empty text.
    - `- <p className="text-center">No items yet! Add one above!</p>`
    - `+ <p className="text-center">You have no todo items yet! Add one above!</p>`
  - Let's build our updated version of the image, using the same command we used before.
    - `$ docker build -t getting-started .`
  - Let's start a new container using the updated code.
    - `$ docker run -dp 3000:3000 getting-started`



Update the application

- Remove a container using the CLI
  - Get the ID of the container by using the docker ps command.
    - `$ docker ps`
  - Use the docker stop command to stop the container.
    - `# Swap out <the-container-id> with the ID from docker ps`
    - `$ docker stop <the-container-id>`
  - Once the container has stopped, you can remove it by using the docker rm command.
    - `$ docker rm <the-container-id>`

- Start the updated app container
  - Now, start your updated app.
    - `$ docker run -dp 3000:3000 getting-started`

Share the application

- To share Docker images, you have to use a Docker registry.
  - The default registry is Docker Hub and is where all of the images we've used have come from.

- Create a repo
  - To push an image, we first need to create a repository on Docker Hub.

- Push the image
  - Use the docker tag command to give the getting-started image a new name. Be sure to swap out YOUR-USER-NAME with your Docker ID.

    ```
    docker tag getting-started YOUR-USER-NAME/getting-started
    ```

  - Now try your push command. If you're copying the value from Docker Hub, you can drop the tagname portion, as we didn't add a tag to the image name. If you don't specify a tag, Docker will use a tag called latest.

    ```
    docker push YOUR-USER-NAME/getting-started
    ```

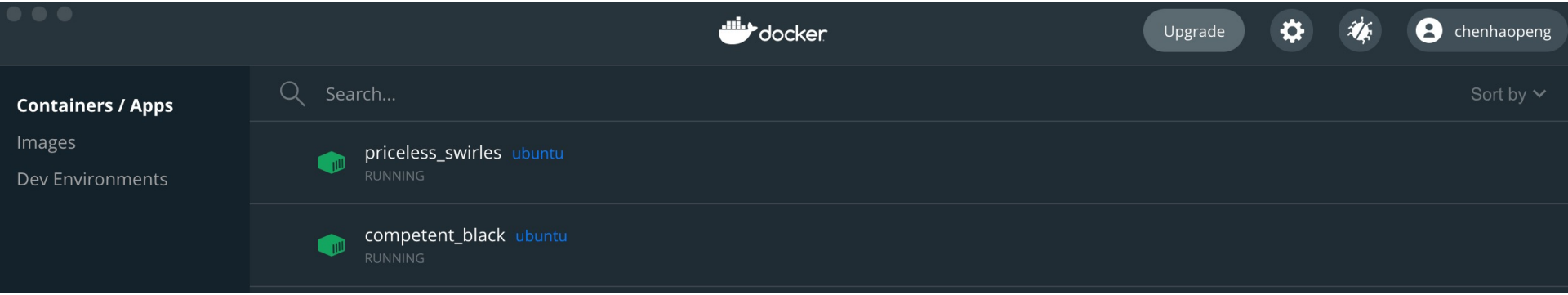
Persist the DB

- The container's filesystem
  - When a container runs, it uses the various layers from an image for its filesystem.
  - Each container also gets its own "scratch space" to create/update/remove files.
  - Any changes won't be seen in another container, *even if* they are using the same image.

- To see this in action, we're going to start two containers and create a file in each.
  - Start an ubuntu container that will create a file named /data.txt with a random number between 1 and 10000.
  - `$ docker run -d ubuntu bash -c "shuf -i 1-10000 -n 1 -o /data.txt && tail -f /dev/null"`
  - Start another ubuntu container.
  - `$ docker run -it ubuntu`

Persist the DB

REin
REliable, INtelligent & Scalable Systems




```
chenhaopeng — com.docker.cli ◂ docker exec -it db8b7ac246b82e1a933...
Last login: Wed Sep 29 09:20:06 on ttys000
docker exec -it db8b7ac246b82e1a9337fb0b3a0479bd80b1c36a8c6ec52a2cc98db4e409f3c0
 /bin/sh

The default interactive shell is now zsh.
To update your account to use zsh, please run `chsh -s /bin/zsh`.
For more details, please visit https://support.apple.com/kb/HT208050.
(base) MacBook-Pro-7:~ chenhaopeng$ docker exec -it db8b7ac246b82e1a9337fb0b3a04
79bd80b1c36a8c6ec52a2cc98db4e409f3c0 /bin/sh
# cat /data.txt
431
# ls
bin   data.txt  etc   lib    lib64   media  opt   root  sbin  sys  usr
boot  dev       home  lib32  libx32  mnt    proc  run   srv   tmp  var
#
```

```
chenhaopeng — com.docker.cli ◂ docker exec -it 47bc474286ec3f1c3432...
Last login: Wed Sep 29 17:25:51 on ttys001
docker exec -it 47bc474286ec3f1c3432605baf0dd84d09522c97d6cb5b76f0dea9ab81f97374
 /bin/sh

The default interactive shell is now zsh.
To update your account to use zsh, please run `chsh -s /bin/zsh`.
For more details, please visit https://support.apple.com/kb/HT208050.
(base) MacBook-Pro-7:~ chenhaopeng$ docker exec -it 47bc474286ec3f1c3432605baf0d
d84d09522c97d6cb5b76f0dea9ab81f97374 /bin/sh
# ls
bin   dev  home  lib32  libx32  mnt   proc  run   srv  tmp  var
boot  etc  lib   lib64  media   opt   root  sbin  sys  usr
#
```

Container Volume

- With the previous experiment, we saw that each container starts from the image definition each time it starts.
  - While containers can create, update, and delete files, those changes are lost when the container is removed and all changes are isolated to that container.
  - With volumes, we can change all of this.
- Volumes provide the ability to connect specific filesystem paths of the container back to the host machine.
  - If a directory in the container is mounted, changes in that directory are also seen on the host machine.
  - If we mount that same directory across container restarts, we'd see the same files.
- There are two main types of volumes.
  - We will eventually use both, but we will start with **named volumes**.

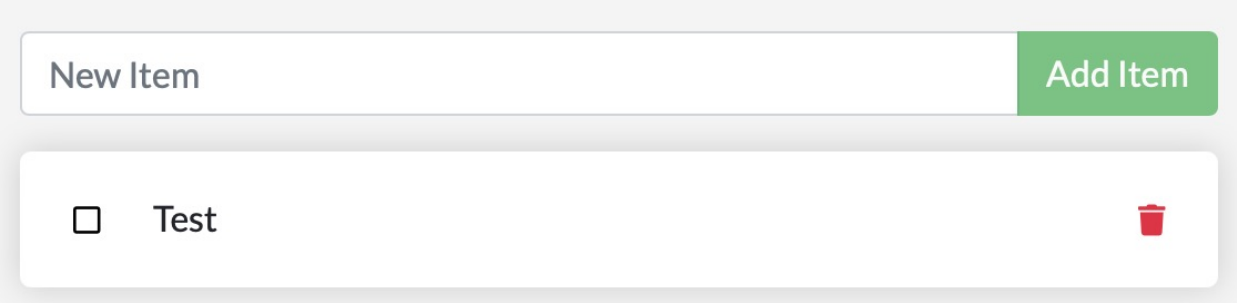
Persist the todo data

- By default, the todo app stores its data in a SQLite Database at /etc/todos/todo.db in the container's filesystem.
  - With the database being a single file, if we can persist that file on the host and make it available to the next container, it should be able to pick up where the last one left off.

- We are going to use a **named volume**.
  - Think of a named volume as simply a bucket of data.
  - Docker maintains the physical location on the disk and you only need to remember the name of the volume.
  - Every time you use the volume, Docker will make sure the correct data is provided.

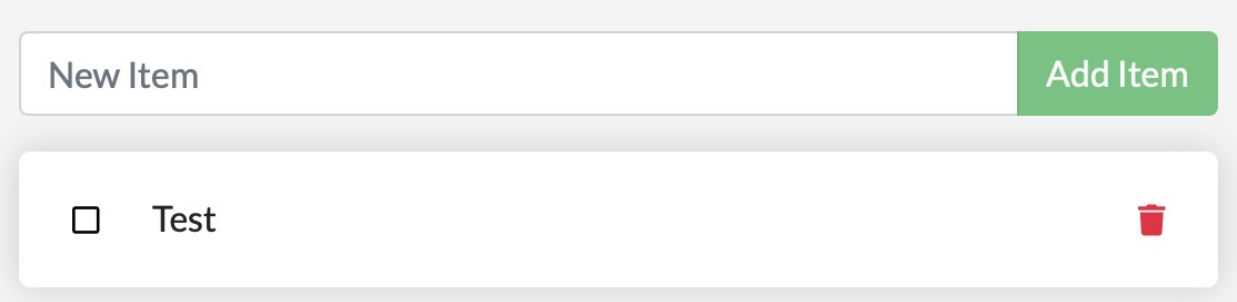
Persist the todo data

1. Create a volume by using the docker volume create command.
   - `$ docker volume create todo-db`
   - Stop and remove the todo app container once again in the Dashboard (or with docker rm -f <id>), as it is still running without using the persistent volume.
2. Start the todo app container, but add the -v flag to specify a volume mount.
   - We will use the named volume and mount it to /etc/todos, which will capture all files created at the path.
   - `$ docker run -dp 3000:3000 -v todo-db:/etc/todos getting-started`
3. Once the container starts up, open the app and add a few items to your todo list.



Persist the todo data

4. Stop and remove the container for the todo app.
   - Use the Dashboard or docker ps to get the ID and then docker rm -f <id> to remove it.
5. Start a new container using the same command from above.
6. Open the app. You should see your items still in your list!
7. Go ahead and remove the container when you're done checking out your list.



Dive into the volume

- A lot of people frequently ask
  - "Where is Docker *actually* storing my data when I use a named volume?"
  - If you want to know, you can use the `docker volume inspect` command.

```
(base) MacBook-Pro-7:~ chenhaopeng$ docker volume inspect todo-db
[
    {
        "CreatedAt": "2021-10-07T09:41:27Z",
        "Driver": "local",
        "Labels": {},
        "Mountpoint": "/var/lib/docker/volumes/todo-db/_data",
        "Name": "todo-db",
        "Options": {},
        "Scope": "local"
    }
]
```

- The Mountpoint is the actual location on the disk where the data is stored.
  - Note that on most machines, you will need to have root access to access this directory from the host. But, that's where it is!

Use bind mounts

- With **bind mounts**, we control the exact mountpoint on the host.
  - We can use this to persist data, but it's often used to provide additional data into containers.
  - When working on an application, we can use a bind mount to mount our source code into the container to let it see code changes, respond, and let us see the changes right away.

- Bind mounts and named volumes are the two main types of volumes that come with the Docker engine.
  - However, additional volume drivers are available to support other uses cases (SFTP, Ceph, NetApp, S3, and more).

Use bind mounts

	Named Volumes	Bind Mounts
Host Location	Docker chooses	You control
Mount Example (using `-v`)	my-volume:/usr/local/data	/path/to/data:/usr/local/data
Populates new volume with container contents	Yes	No
Supports Volume Drivers	Yes	No

Use bind mounts

- Start a dev-mode container
  - To run our container to support a development workflow, we will do the following:
  - Mount our source code into the container
  - Install all dependencies, including the “dev” dependencies
  - Start nodemon to watch for filesystem changes

1. Make sure you don’t have any previous getting-started containers running.
2. Run the following command in the directory of getting-started-master/app

```
$ docker run -dp 3000:3000 \
    -w /app -v "$(pwd):/app" \
    node:12-alpine \
    sh -c "yarn install && yarn run dev"
```

Use bind mounts

2. Run the following command in the directory of getting-started-master/app

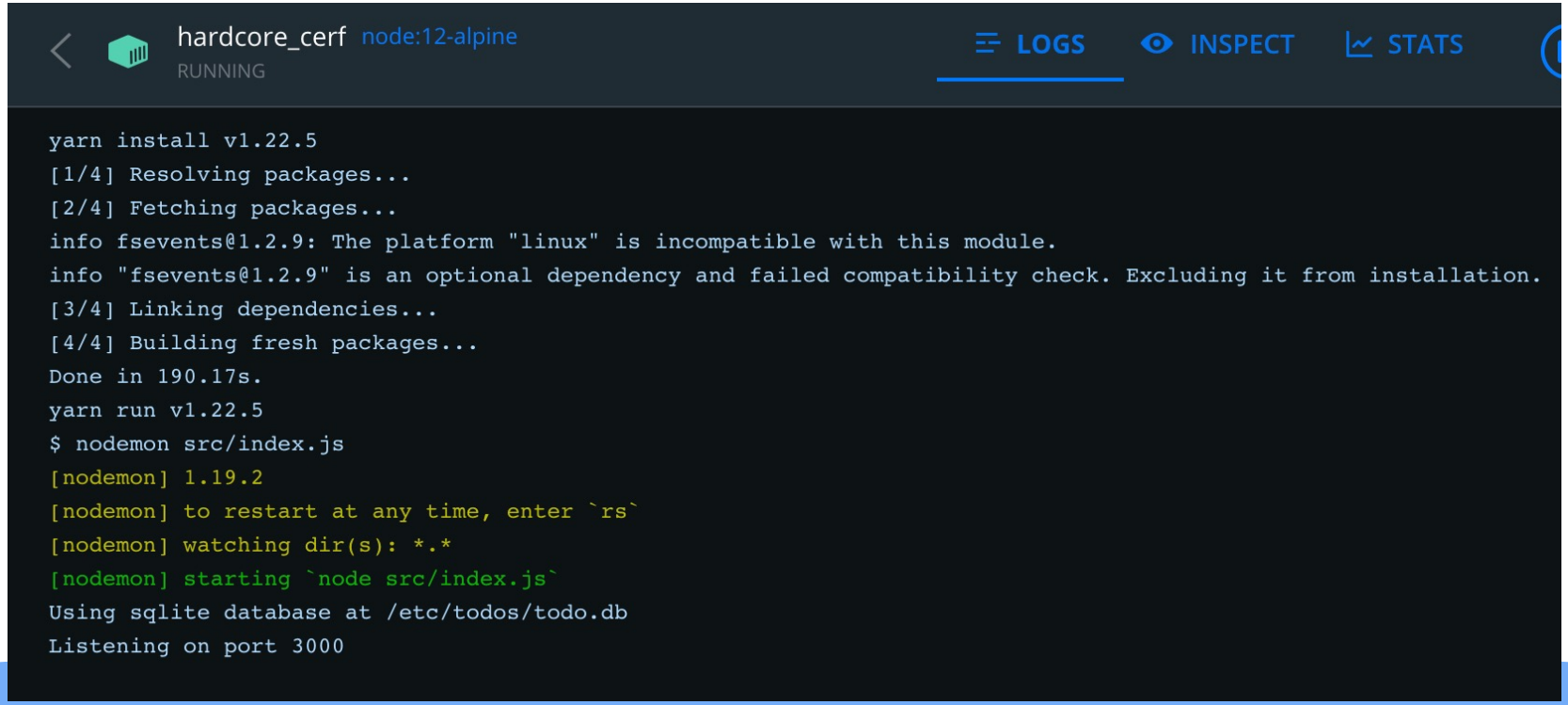
```
$ docker run -dp 3000:3000 \
    -w /app -v "$(pwd):/app" \
    node:12-alpine \
    sh -c "yarn install && yarn run dev”
```

- -dp 3000:3000 - same as before. Run in detached (background) mode and create a port mapping
- -w /app - sets the “working directory” or the current directory that the command will run from
- -v "$(pwd):/app" - bind mount the current directory from the host in the container into the /app directory
- node:12-alpine - the image to use. Note that this is the base image for our app from the Dockerfile
- sh -c "yarn install && yarn run dev" - the command. We’re starting a shell using sh (alpine doesn’t have bash) and running yarn install to install *all* dependencies and then running yarn run dev. If we look in the package.json, we’ll see that the dev script is starting nodemon.

Use bind mounts

REliable, INtelligent & Scalable Systems

3. You can watch the logs using docker logs -f <container-id>. You'll know you're ready to go when you see this:

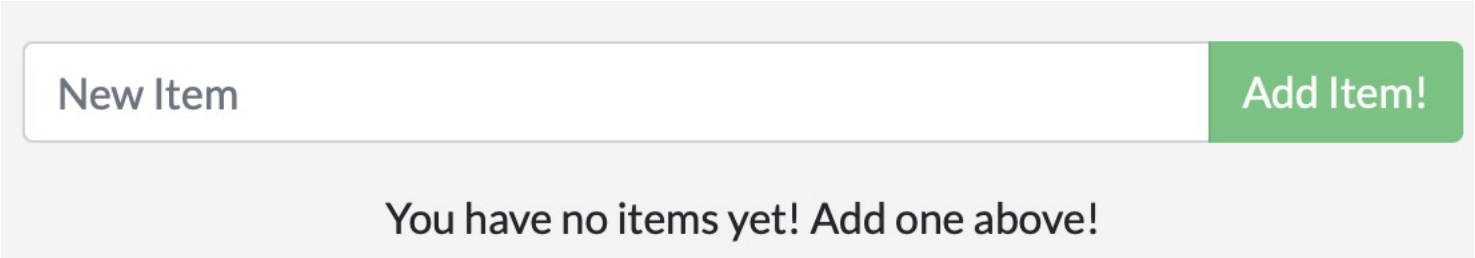


Use bind mounts

4. Now, let's make a change to the app. In the src/static/js/app.js file, let's change the "Add Item" button to simply say "Add".

```
- {submitting ? 'Adding...' : 'Add Item'}
+ {submitting ? ‘Adding...’ : ‘Add Item!'}
```

5. Simply refresh the page (or open it) and you should see the change reflected in the browser almost immediately.
   - It might take a few seconds for the Node server to restart, so if you get an error, just try refreshing after a few seconds.



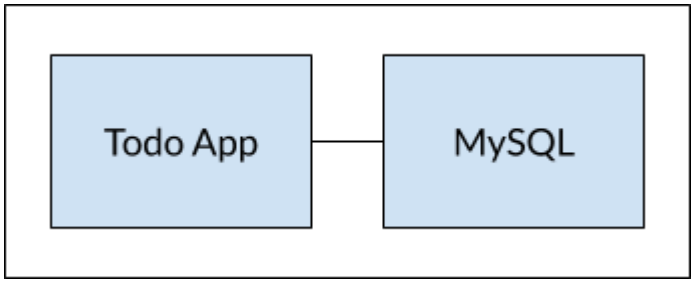
Use bind mounts

6. Feel free to make any other changes you'd like to make.
   - When you're done, stop the container and build your new image using
   - `docker build -t getting-started .`

- Using bind mounts is *very* common for local development setups.
  - The advantage is that the dev machine doesn't need to have all of the build tools and environments installed.
  - With a single docker run command, the dev environment is pulled and ready to go.

Multi container apps

- In general, each container should do one thing and do it well. A few reasons:
  - There's a good chance you'd have to scale APIs and front-ends differently than databases
  - Separate containers let you version and update versions in isolation
  - While you may use a container for the database locally, you may want to use a managed service for the database in production. You don't want to ship your database engine with your app then.
  - Running multiple processes will require a process manager (the container only starts one process), which adds complexity to container startup/shutdown



Multi container apps

- Container networking
  - Remember that containers, by default, run in isolation and don't know anything about other processes or containers on the same machine. So, how do we allow one container to talk to another?
  - The answer is **networking**.
- Note
  - If two containers are on the same network, they can talk to each other.
  - If they aren't, they can't.
- Start MySQL
  - There are two ways to put a container on a network:
  - 1) Assign it at start or
  - 2) connect an existing container.
  - For now, we will create the network first and attach the MySQL container at startup.

Multi container apps

- Start MySQL
  - Create the network.

```
docker network create todo-app
```

  - Start a MySQL container and attach it to the network. We're also going to define a few environment variables that the database will use to initialize the database.

```
docker run -d \
 --network todo-app --network-alias mysql \
 -v todo-mysql-data:/var/lib/mysql \
 -e MYSQL_ROOT_PASSWORD=secret \
 -e MYSQL_DATABASE=todos \
 mysql:5.7
```

Multi container apps

- Start MySQL
  - To confirm we have the database up and running, connect to the database and verify it connects.

```
docker exec -it <mysql-container-id> mysql -u root -p
```

  - When the password prompt comes up, type in **secret**. In the MySQL shell, list the databases and verify you see the todos database.

```
mysql> SHOW DATABASES;
```

  - You should see output that looks like this:

```
+--------------------+
| Database           |
+--------------------+
| information_schema |
| mysql              |
| performance_schema |
| sys                |
| todos              |
+--------------------+
5 rows in set (0.00 sec)
```

Multi container apps

- Connect to MySQL
  - We're going to make use of the nicolaka/netshoot container,
    - which ships with a *lot* of tools that are useful for troubleshooting or debugging networking issues.
  - Start a new container using the nicolaka/netshoot image. Make sure to connect it to the same network.

    ```
    docker run -it --network todo-app nicolaka/netshoot
    ```

  - Inside the container, we're going to use the dig command, which is a useful DNS tool.
  - We're going to look up the IP address for the hostname mysql.

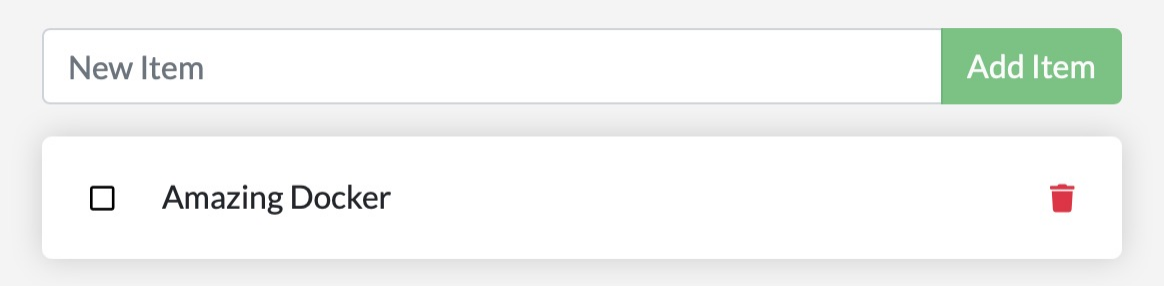
    ```
    dig mysql
    ```

Multi container apps

- Run your app with MySQL
  - We'll specify each of the environment variables above, as well as connect the container to our app network.

```
docker run -dp 3000:3000 -w /app -v "$(pwd):/app" --network todo-app -e
MYSQL_HOST=mysql -e MYSQL_USER=root -e MYSQL_PASSWORD=secret -e
MYSQL_DB=todos node:12-alpine sh -c "yarn install && yarn run dev"
```

  - Open the app in your browser and add a few items to your todo list.



Multi container apps

- Run your app with MySQL
  - Connect to the mysql database and prove that the items are being written to the database. Remember, the password is **secret**.

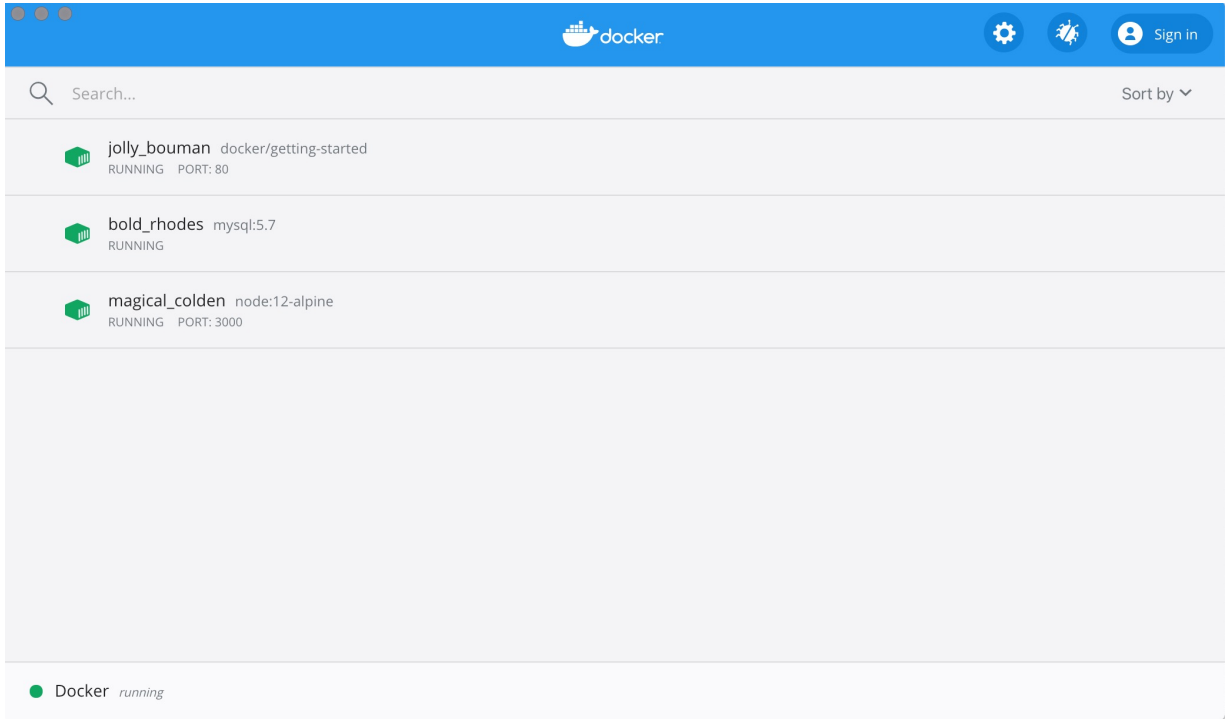
```
docker exec -it <mysql-container-id> mysql -p todos
```

  - And in the mysql shell, run the following:

```
mysql> select * from todo_items;
+--------------------------------------+----------------+-----------+
| id                                   | name           | completed |
+--------------------------------------+----------------+-----------+
| caab1b30-ea98-4382-8c19-bae658e88185 | Amazing Docker |         0 |
+--------------------------------------+----------------+-----------+
1 row in set (0.00 sec)
```

Multi container apps

- Run your app with MySQL



Use Docker Compose

- Docker Compose is a tool that was developed to help define and share multi-container applications.
  - With Compose, we can create a YAML file to define the services and with a single command, can spin everything up or tear it all down.

- The *big* advantage of using Compose is
  - you can define your application stack in a file, keep it at the root of your project repo (it's now version controlled), and easily enable someone else to contribute to your project.
  - Someone would only need to clone your repo and start the compose app. In fact, you might see quite a few projects on GitHub/GitLab doing exactly this now.

Use Docker Compose

- Create the Compose file
  - At the root of the app project, create a file named docker-compose.yml.

- Define the app service

```
version: "3.7"
services:
 app:
 image: node:12-alpine
 command: sh -c "yarn install && yarn run dev"
 ports:
   - 3000:3000
 working_dir: /app
 volumes:
   - ./:/app
 environment:
  MYSQL_HOST: mysql
  MYSQL_USER: root
  MYSQL_PASSWORD: secret
  MYSQL_DB: todos
```

Use Docker Compose

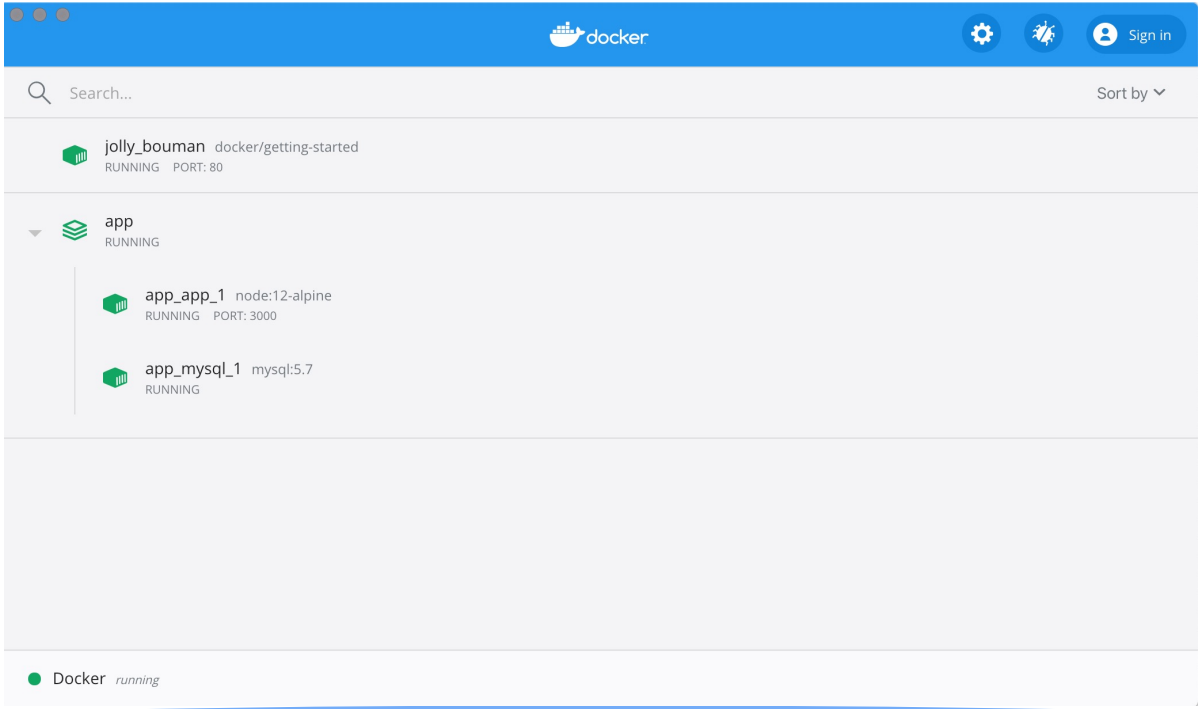
- Define the MySQL service

```
version: "3.7"
services:
  app:
    # The app service definition
  mysql:
    image: mysql:5.7
    volumes:
      - todo-mysql-data:/var/lib/mysql
    environment:
      MYSQL_ROOT_PASSWORD: secret
      MYSQL_DATABASE: todos
volumes:
  todo-mysql-data:
```

Use Docker Compose

- Run the application stack

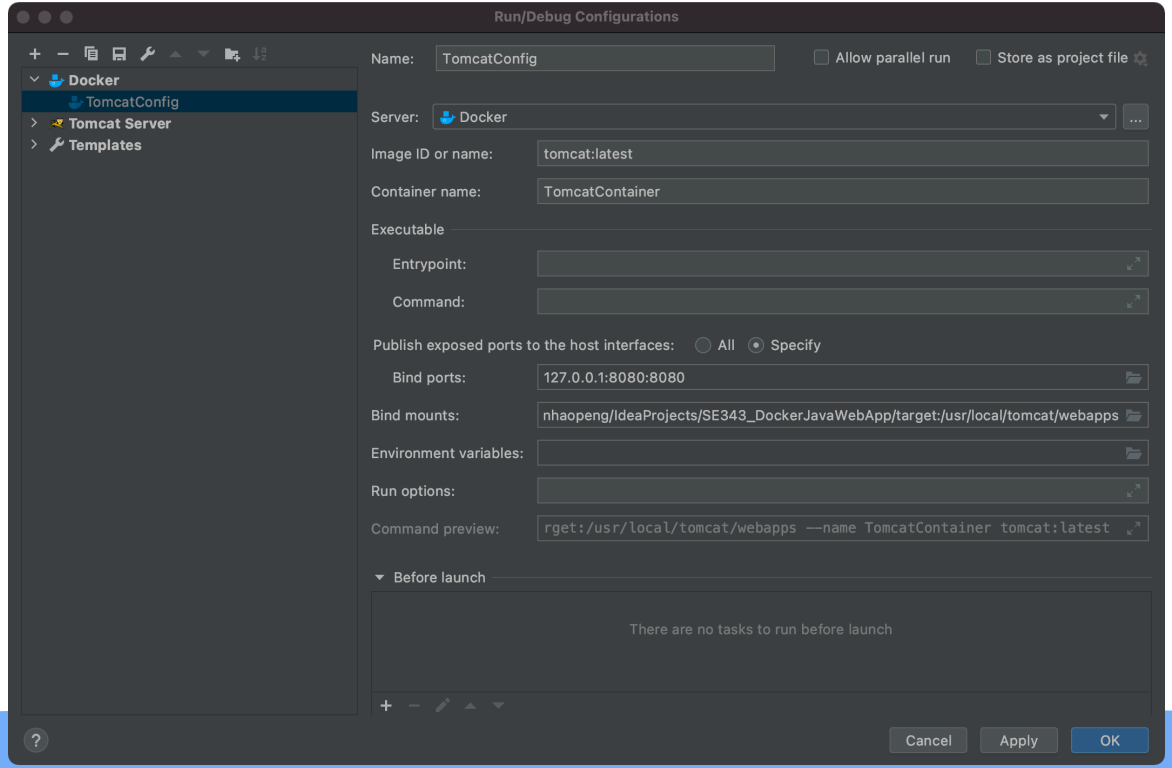
```
docker-compose up -d
```



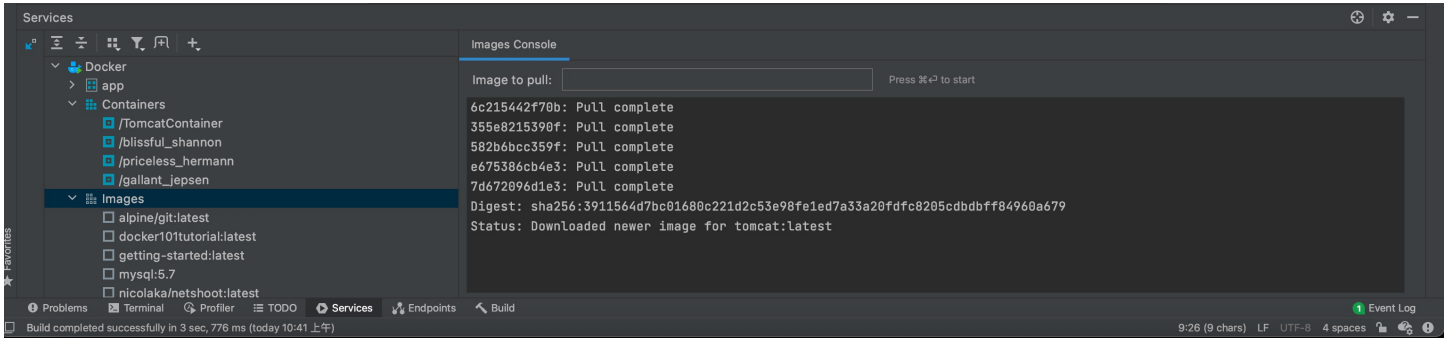
Deploy a Java web application inside a Tomcat server container in IDEA

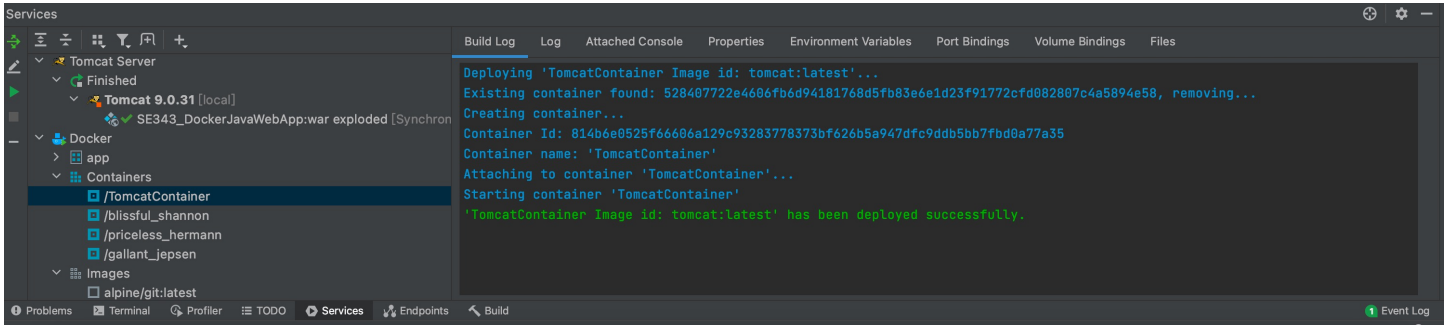
REin
REliable, INtelligent & Scalable Systems

- *From:*
  - https://www.jetbrains.com/help/idea/deploying-a-web-app-into-an-app-server-container.html



Deploy a Java web application inside a Tomcat server container in IDEA





- http://127.0.0.1:8080/SE3353_22_DockerJavaWebApp/

作业十

- 请你根据上课内容，针对你在E-BookStore项目中的数据库设计，完成下列任务：
  1. 请你参照课程样例，构建你的E-BookStore的集群，它应该至少包含 1 个nginx实例(负载均衡) + 1 个Redis实例(存储session) + 2 个Tomcat实例 + 2个MySQL实例(主从备份)。(4分)
  2. 所使用的框架不限，例如可以不使用nginx而选用其他负载均衡器，或不使用Redis而选用其他缓存工具。
  3. 参照上课演示的案例，将上述系统实现容器化部署，即负载均衡器、缓存、注册中心和服务集群都在容器中部署。(1分)

  - 请提交一份Word文档，详细叙述你的实现方式；并提交你的工程代码。

- 评分标准：
  - 能够正确地部署和运行上述系统，在迭代验收时需当面演示。
  - 部署方案不满足条件或无法正确运行，则视情况扣分。

Thank You!