Containerization with Docker

- ***Containerization*** is a modern computing paradigm in which application software is run within an environment that is virtualized at the operating system level.

- In particular, in a containerized system, software runs in ***containers***, which are independent, isolated userspace instances, i.e. virtualized environments that run on a physical host machine.

- While containers are like virtual machines in some ways, providing resource isolation, portability, etc., they are distinguished by being virtualized at the operating system level instead of at the hardware level.

- In other words, a container does not need its own copy of the operating system. Instead, it makes use of modern features of its host machine's underlying operating system to run directly on that host's OS kernel. Indeed, a running container is simply a discrete process, just like any other executable.

- Because containers, unlike VMs, do not need their own copy of the OS, they are much more lightweight than VMs, while still allowing software to be isolated from the physical infrastructure on which it runs.

- This feature makes containers extremely portable. They can be used to wrap application software along with all of its runtime dependencies into a lightweight package, and this package can be moved from machine to machine, running pretty much the same way everywhere. The only requirement is that each underlying host machine must support containerization.

- Because of this portability, containers are extremely useful in software development, since it allows different developers and teams running different development environments to collaborate on a single piece of software, using one or more containers to encapsulate that software's runtime dependencies.

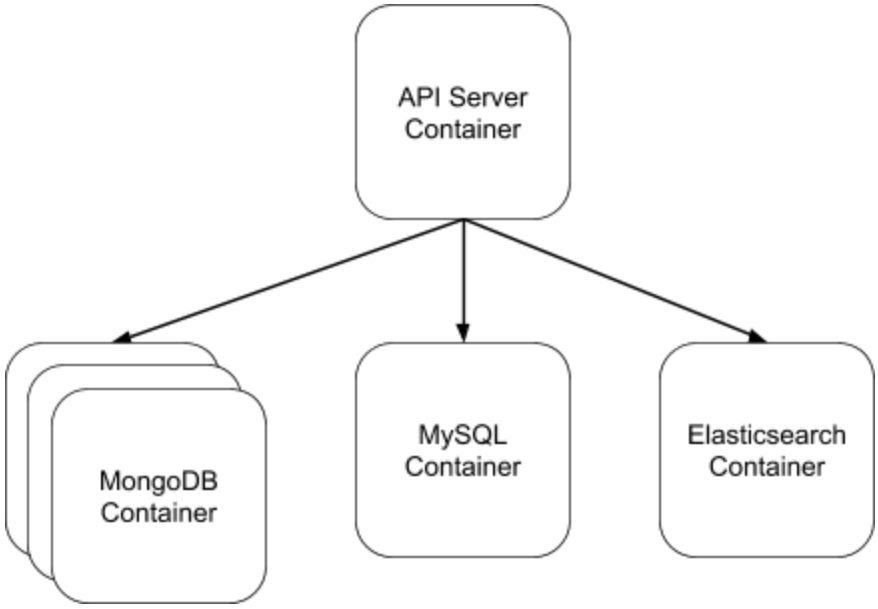
- Perhaps more importantly, containers make software *deployment* very easy as well. If an application is containerized along with its dependencies, it can be deployed to any infrastructure that supports containerization, and scaling the application can often be done by simply running more containers.

- As an example of how this might work, consider implementing a RESTful API built with Node.js/Express stack and using both MongoDB and MySQL as underlying application data stores and Elasticsearch to provide full-text search.

- A potential containerization of this application would include four different containers:
    - One to run the API server with Node.js/Express
    - One to run MongoDB
    - One to run MySQL
    - One to run Elasticsearch

- The API server (in its container) would communicate with the services in the other three containers as needed.

- In this setup, application developers could simply run their own MongoDB, MySQL, and Elasticsearch containers for application testing, and the application could be deployed with a set of production containers running these services. To scale MongoDB, more MongoDB containers could be used.



- The benefits of containerization have made it a popular approach to developing modern system architectures, and many organizations are moving to containerize their applications and services (or have already done so).

Docker

- ***Docker*** is an open-source container platform. It is probably today's most popular way to containerize software.

- There are several tools that are part of the Docker platform, but the most essential one is a tool for simply creating and running containers. We'll use this tool extensively in this course, along with another tool called Docker Compose, which is a tool for defining and running multi-container applications.

- Before we move to discuss using Docker, a brief note on installing Docker.

Installing Docker

- You'll need to make sure you have Docker and Docker Compose installed on your laptop for this course.

- There is a native version of Docker available for Linux and for newer versions of MacOS and Windows. In this course, we will specifically be using Docker CE (Community Edition). You can find all of the Docker releases on this page: https://docs.docker.com/install/.
    - The Docker installation page lists Linux releases of Docker CE for Ubuntu, Debian, CentOS, and Fedora. Other Linux distributions may also have Docker available through their package managers. It is also possible to download Docker binaries directly from here: https://docs.docker.com/install/linux/docker-ce/binaries/.

- Note that for Windows and Mac installations, your machine and OS must meet the requirements listed on the respective Docker installation page for your OS. If your machine or OS does not meet these requirements, an alternative method for installing Docker is to use Docker Toolbox, which uses a tool called Docker Machine to run Docker on a VirtualBox VM. This approach should work on any machine that supports VirtualBox. You can find more information about Docker Toolbox here: https://docs.docker.com/toolbox/overview/.
    - Note that running Docker via Docker Toolbox may require just a few extra steps to make sure your VM is running each time you want to use Docker. The Docker Toolbox page linked above contains links to documentation that describes how to do this.

Docker images and containers

- Once you have Docker installed, you're ready to start exploring how to use it.

- When using Docker, It's important to understand that there are actually two different ways in which a container manifests itself in Docker: an ***image*** and a ***container***.

- A Docker image is an executable package that contains an application and its runtime dependencies (code, libraries, environment variables, configuration and data files, etc.). Though this isn't strictly the case, it is useful to think of a Docker image as a file that specifies what's in a container and what happens when the container is run.
    - You can see all of the Docker images on your machine by running the command `docker image ls`.

- A Docker container, by contrast, is a runtime instantiation of an image, i.e. an image that's been executed (which results in an in-memory process). A Docker container can be in a running state, but it can also be paused or stopped.
    - You can see all of the Docker containers (running, paused, and stopped) on your machine by running the command `docker container ls`.

- Just like many processes can be created at the same time from a single executable file, many Docker containers can be created at the same time from a single Docker image. Each running container can evolve in its own way, depending on how you interact with it.

- We can start to see the difference between images and containers by running the Docker hello-world image:

  ```
  docker run hello-world
  ```

- When you run this command for the first time, you'll first see some status messages that look like this:

```
Unable to find image 'hello-world:latest' locally
latest: Pulling from library/hello-world
9bb5a5d4561a: Pull complete
Digest: sha256:f5233545e435...
Status: Downloaded newer image for hello-world:latest
```

- These messages are telling you that Docker can't find the hello-world image locally when it tries to, so it downloads it from a public ***Docker registry*** called Docker Hub. We'll talk more about Docker Hub later, but for now here's what's important to know:
    - Docker Hub is an open registry of publicly-available pre-built Docker images.
    - If you try to run a container from a Docker image that's not on your machine, Docker will automatically check to see if that image exists on Docker Hub, and, if it is, it will download the image for you.

- In fact, if you run `docker image ls` now, you'll see that you now have the hello-world image on your machine, e.g.:

```
$ docker image ls
REPOSITORY    TAG     IMAGE ID        CREATED      SIZE
hello-world   latest  e38bc07ac18e    9 days ago   1.85kB
```

- You can also see the hello-world container you just executed by running `docker container ls` (with the `--all` option, since the container is stopped):

```
$ docker container ls --all
CONTAINER ID   IMAGE        COMMAND    CREATED         ...
99d9b6b447b4   hello-world  "/hello"   10 seconds ago ...
```

- So what happened here when we ran a container from the hello-world image? The output of the hello-world image gives us a pretty in-depth picture of everything that took place in this process. Note in particular the following line:

```
3. The Docker daemon created a new container from that
image [hello-world] which runs the executable that produces
the output you are currently reading.
```

- Notice specifically that the execution sequence looks like this:
    - The Docker daemon creates a container from the hello-world image.
    - The container runs an executable to produce the output you see.

- In other words, the executable that produced the output you see in your terminal lives *in the container*, and it is the *container* that runs that executable. Any runtime dependencies for the executable also live in the container. For example, if the executable was compiled from C code with a different version of the C library than is installed on the host machine, that's OK. The container has that version of the C library installed for the executable to use.

A more in-depth Docker run

- Let's do a more involved container run to explore some of Docker's capabilities. This time, we'll run a container from the Ubuntu image on Docker Hub.

- The `docker run` command is the Docker command used in the terminal to run a new container. The general form of this command is:

```
docker run [OPTIONS] IMAGE[:TAG|@DIGEST] [COMMAND] [ARG...]
```

- Below is the command we'll run to start our container. This command will set us up to do some more advanced things with the container:

```
docker run -it                \
     --name test-container    \
     -e "PORT=8080"           \
     -v "$(pwd)":/home/test   \
     -p 34567:8080            \
     ubuntu:latest            \
          /bin/bash
```

- Let's look at the individual pieces of this command separately:
    - `-it` – this combines two options to the `docker` command, `-i` and `-t`, which together allow us to run an interactive terminal on the container.
    - `--name test-container` – this option applies a name to our container. We'll see this name if we run `docker ps`.

    - `-e "PORT=8080"` – this option sets the environment variable `PORT` to have the value 8080 on the container. In general, we can set any number of environment variables on the container by passing multiple `-e` options.
    - `-v "$(pwd)":/home/test` – this option "mounts" the current working directory on the host machine (i.e. the output of the `pwd` command). The contents of this host-machine directory will be available in the directory `/home/test` in the container. Anything created in this directory on the container will also be available on the host machine.
    - `-p 34567:8080` – this option publishes a specified port on the container (8080) to a specified port on the host machine (34567). In other words, by specifying this option, network requests made to port 34567 on the host machine will be sent to port 8080 on the container.
    - `ubuntu:latest` – this argument specifies the Docker image upon which our container will be based. Specifically, we'll use the version of the image named `ubuntu` and tagged `latest`.
    - `/bin/bash` – this specifies the command our container will execute. Specifically, our container will execute a Bash shell. The `-it` options we specified above will allow us to interact with this shell.
        - Note that we did not specify a command to execute when we ran the hello-world image. This is because the hello-world image is set to run a default command.

- After running this `docker run` command you'll see that you're now in a Bash shell, which is running *in the container*. The Ubuntu image from which we created this container gives us a complete filesystem, which you can see if you run the `ls` command:

```
# ls
bin   dev  home  lib64  mnt  proc  run   srv  tmp  var
boot  etc  lib   media  opt  root  sbin  sys  usr
```

- We can navigate around this filesystem using cd, just like we would in any other terminal. For example, we can navigate to `/home/test`, which is the container directory to which we mounted the host-machine directory from which we launched the container:

```
cd /home/test
```

- If you run `ls` here, you'll see whatever were the contents of your host-machine directory.

- Now that we're getting comfortable in our container, let's start to see what we can do here. Remember, our container is very much like a virtual machine. We can create files on our container. For example, let's install some software (cURL) using Ubuntu's package manager, `apt`:

```
apt-get update
apt-get install curl
```

- With cURL installed, we can also install Node.js:

```
curl -sL https://deb.nodesource.com/setup_9.x | bash -
apt-get install nodejs
```

- With Node.js installed, let's write and run a simple server. We'll first install Express:

```
npm install express
```

- Remember, the `/home/test` directory on the container is actually a directory mounted from the host machine, specifically, the directory from which we launched the container. If you navigate to that directory *on your host machine*, you should see a `node_modules/` directory there with Express installed. This was created from inside the container.

- Open a file called `test.js` in this host directory with your favorite editor. We'll write a simple server here:

```
const express =      require('express');
const app = express();
const port = process.env.PORT || 8000;

app.get('/', function (req, res) {
  console.log("== Got a request!");
  res.status(200).send("Hello from Docker!\n");
});
```

```
app.listen(port, function () {
  console.log("== Server running on port", port);
});
```

- Now, if you launch your server, you'll see that it runs on port 8080, the value we specified for the PORT environment variable when we launched the Docker container, instead of the default 8000:

```
# node test.js
== Server running on port 8080
```

- Using Postman, cURL, or another tool, send a GET request to the root URL path on port 34567 of the host machine. E.g. from a terminal on the *host* machine:

```
$ curl "http://localhost:34567"
Hello from Docker!
```

- Here, you can see the response back from the server running *in the container*. This means our host-machine port 34567 is successfully being forwarded to the server port 8080 on our container. If you check the container terminal, you'll see the request we just made logged there:

```
== Got a request!
```

- Our setup is working! We're running software in a Docker container and interacting with it from the outside world. We've containerized some software. In particular, the runtime dependencies (e.g. Node.js) for our server were all installed in the container, not on our host machine. Our host machine did not need Node.js installed.

- Of course, the process we went through to get our server running was not very amenable to replication and portability, since we did a lot of manual setup work to get Node.js installed on our container. Luckily, there's a better way to make portable Docker images that package up a specific piece of software to execute.

Creating your own Docker image with a Dockerfile

- Docker images are great because they're very portable, and for much of the software we'll want to run, we'll be able to find a pre-built image on Docker Hub that we can download and use right out of the box, for the most part. Often,

though, you'll need to do something these images don't quite handle.

- To handle such cases, you can build your own Docker image. We do this by writing a specification of the image we want to build in a Dockerfile.

- A Dockerfile is a set of instructions for building a Docker image. There are many kinds of instructions we can specify in a Dockerfile. For example, we could specify terminal commands to be run within the image to perform setup steps (e.g. installing software), or we could copy files or directories from the machine on which we're building the image into the image.

- To demonstrate how Dockerfiles work, let's write one to package up the API server we wrote previously for an Airbnb-like application. This is the code we'll use as our starting point (note that a specific commit is specified here):

  https://github.com/OSU-CS493-Sp18/docker/tree/3f1c3708820c0bad03b2e7002be0da6de7aad51c

- In order to containerize a piece of software, it's common to place a Dockerfile specifying that software's runtime environment at the top level of the software's source code repository. We'll do that for our API server, creating a file named `Dockerfile` at the top level of our git repo.

- In fact, every Docker image is actually composed of a collection of ***layers***, and each instruction in a Dockerfile adds a new layer to the image.

- To give the Docker image builder a starting point onto which to start placing new layers, every Docker image built with a Dockerfile must be based on an already-existing Docker image. New layers created by the Dockerfile are placed on top of the last layer in this base image, which must be specified first in the Dockerfile using the `FROM` instruction.

- In this case, we will base our API server's Docker image off of the official Node.js image (version 9), which has Node.js and its dependencies already installed:

  ```
  FROM node:9
  ```

- Next, we'll set the working directory for the subsequent instructions in the Dockerfile using the `WORKDIR` instruction. As will become clear, this will be the

directory in which we'll install and run our server, so we'll use a directory dedicated to that purpose:

```
WORKDIR /usr/src/app
```

- Next, we'll begin the work of installing our server's runtime dependencies. Since we have our server's dependencies specified in `package.json` (and `package-lock.json`), we'll be able to install them by running `npm install`. We just have to make sure `package.json` and `package-lock.json` are present in the image. To do this, we can use the `COPY` instruction to copy those files from our local directory into the directory on the image we specified with the `WORKDIR` instruction above:

  ```
  COPY package*.json ./
  ```

- With those files copied to the image, we can run `npm install` to install our server's dependencies. Do do this, we'll use the `RUN` instruction, which can execute arbitrary terminal commands within the Docker image being built:

  ```
  RUN npm install
  ```

- With our server's dependencies installed in the Docker image, we're now ready to put our server code there, too. We can use another `COPY` instruction to do this, copying the entire contents of the server's code directory to the working directory on the image:

  ```
  COPY . .
  ```

- Now we can set the image up to run our server. The first thing we'll do is work on exposing the port on which we'll run the server. Remember that our server is set up to run on the port specified by the `PORT` environment variable. We can explicitly set the value of this environment variable using an ENV instruction:

  ```
  ENV PORT=8000
  ```

- Then, we can use the value of the `PORT` environment variable in an `EXPOSE` instruction to indicate that containers based on our image will listen on the specified port:

```
EXPOSE ${PORT}
```

- Finally, we can use a `CMD` instruction to set the default command to be run when executing a container from this image. Specifically, we'll want the container to run our API server by executing `npm start`:

  ```
  CMD [ "npm", "start" ]
  ```

  - Note here that each individual token of the command to run is specified as an element in an array here.

- Importantly, the command specified with a `CMD` instruction is executed from within the working directory specified by the `WORKDIR` instruction. This is what we want, since it corresponds to the directory where we installed our server's source code and dependencies.

- One last thing we'll want to do is put a `.dockerignore` file in place. This file will, among other things, allow us to specify files to be ignored when copying files to our image with `COPY` instructions in our Dockerfile. We specifically want to make sure that the `node_modules/` directory is not copied from our local machine onto the image, so we'll add that to our `.dockerignore` file, along with the `.git` directory and any debug logs generated locally:

  ```
  .git
  node_modules
  npm-debug.log
  ```

- Now we're ready to build an image from our Dockerfile. To do this, we'll run the `docker build` command from our terminal:

  ```
  docker build -t book-a-place-api .
  ```

- Two things to note about the `docker build` command above:
  - We're using the `-t` option to tag the image with the name `book-a-place-api`. When we run `docker images`, our image will show up under this name.
  - The `.` argument sets the ***build context*** for our image construction to the current working directory on our local machine. The build context is the set of files available to Docker as it's building an image. For example,

`COPY` instructions copy files from the build context to the image. The Dockerfile is also located at the top level of the build context, by default.

- After executing this command, we'll see Docker process through the instructions in our Dockerfile. At the end, we'll be able to see our newly-built image when we run `docker image ls`:

```
$ docker image ls
REPOSITORY         TAG      IMAGE ID       ...
book-a-place-api   latest   b4942f8a7c44   ...
```

- We're now ready to run a container from this image. The image itself already has our server code and its dependencies packaged up within it. All we need to do is map a host-machine port to the container port (8000) we exposed in the Dockerfile. In this case, we'll map port 8000 on the host machine to this port. We'll also pass the `-d` option to `docker run` to run it in "detached mode", i.e. in the background:

```
docker run -d                 \
     --name book-a-place-api  \
     -p 8000:8000             \
     book-a-place-api
```

- Now, we can again use cURL, Postman, etc. to communicate with our containerized server. For example, this command should successfully fetch the first page of lodgings:

```
curl "http://localhost:8000/lodgings"
```

- Since our container is running in detached mode, we can't see our server's output in the terminal. We can use the command `docker container logs` to access this:

```
$ docker container logs book-a-place-api

> book-a-place@1.1.0 start /usr/src/app
> node server.js

== Server is running on port 8000
== Got request:
```

```
-- URL: /lodgings
-- method: GET
```

Docker networks and communication between containers

- In many situations, we will want to run multi-container applications, where each container runs a various sub-service of the larger application. In these situations, it will be important for containers to be able to communicate with each other. We can achieve this with Docker networks.

- We'll start by exploring how to use ***bridge networks*** to enable containers running on the same host machine to communicate with each other.

- In the context of Docker, a bridge network is a software-based network connecting multiple containers on the same host machine. Containers connected to the same bridged network can communicate with each other, while containers on different bridged networks cannot.

- We'll start out our exploration by simply listing all of the Docker networks that currently exist on our host machine:

```
$ docker network ls
NETWORK ID          NAME                DRIVER              SCOPE
94b6bdfcb4f5        bridge              bridge              local
8448b78ab90b        host                host                local
7011e0a57d01        none                null                local
```

- These are the three default Docker networks you see above:
    - `bridge` is the default bridge network. All containers are connected to this network by default. It allows communication to the outside world as well as between containers connected to it.
    - `host` is used to connect a container directly to the host machine's networking stack.
    - `none` is used to start a container with no networking.

- We are going to create our own bridge network. To do this, we'll use the `docker network create` command. We'll name our network `test-net`, and we'll use

the `bridge` driver to create a bridge network:

```
docker network create --driver bridge test-net
```

- If we used the `docker network ls` command now, we would see our newly-created network listed. We could also run `docker network inspect` to get information about our new network, such as its IP address subnet and gateway IP:

```
docker network inspect test-net
```

- Let's now create two different containers from the Alpine linux image (Alpine is a lightweight Linux distribution). We'll have these containers run the Ash shell (which is Alpine's default) and attach them to this network. Note that we'll start these containers in detached mode using the `-d` option:

```
docker run -dit --name test1 --network test-net alpine ash
docker run -dit --name test2 --network test-net alpine ash
```

- Here, we created one container named `test1` and one named `test2`. We should see them running if we run `docker container ls`. If we used `docker network inspect` to inspect our `test-net` network, we'd see these two containers attached to it.

- Now, let's use the `docker container attach` command to attach to the running `test1` container:

```
docker container attach test1
```

- We should now be in a shell in our container. We can use the `ping` command to demonstrate that this container can communicate over the network to the `test2` container. In fact, because we're using a custom-made bridge network (not the default), we can communicate by container name, since non-default bridge networks can resolve container names to IP addresses:

```
# ping -c 3 test2
PING test2 (172.18.0.3): 56 data bytes
64 bytes from 172.18.0.3: seq=0 ttl=64 time=0.276 ms
64 bytes from 172.18.0.3: seq=1 ttl=64 time=0.185 ms
64 bytes from 172.18.0.3: seq=2 ttl=64 time=0.171 ms

--- test2 ping statistics –
3 packets transmitted, 3 packets received, 0% packet loss
round-trip min/avg/max = 0.171/0.210/0.276 ms
```

- We can hit CTRL-P, CTRL-Q to detach from this container without stopping it, bringing us back to a terminal on our host machine. From there, we can attach to the `test2` container and perform the same test:

```
# ping -c3 test1
PING test1 (172.18.0.2): 56 data bytes
64 bytes from 172.18.0.2: seq=0 ttl=64 time=0.195 ms
64 bytes from 172.18.0.2: seq=1 ttl=64 time=0.185 ms
64 bytes from 172.18.0.2: seq=2 ttl=64 time=0.189 ms

--- test1 ping statistics –
3 packets transmitted, 3 packets received, 0% packet loss
round-trip min/avg/max = 0.185/0.189/0.195 ms
```

Multi-container applications

- Now that we know how to communicate between containers over a Docker network, let's explore how to construct a multi-container application.

- To do this, we'll write a simple relay server. This server will simply listen for requests on a given path. Each time it gets a request, it will generate a single request to our Book-a-Place API server, and it will respond to its own original request by simply sending the response from the Book-a-Place API. In other words, it simply relays requests to the Book-a-Place API.

- Below is the code for the relay server. It simply forwards requests on the path `/` to the Book-a-Place API path `/lodgings` Note that it uses a package called

SuperAgent to perform HTTP GET requests to the Book-a-Place server:

```
const request = require('superagent');
const express = require('express');
const app = express();
const port = process.env.PORT || 8001;

const relayHost = process.env.RELAY_HOST || 'localhost';
const relayPort = process.env.RELAY_PORT || 8000;
const relayPath = process.env.RELAY_PATH || '/lodgings';
const relayURL =
  `http://${relayHost}:${relayPort}${relayPath}`;

app.get('/', function (req, res) {
  request
    .get(relayURL)
    .end(function (err, relayRes) {
      if (err) {
        res.status(500).send(err);
      } else {
        res.status(200).send(relayRes.body);
      }
    });
});

app.listen(port, function() {
  console.log("== Relaying", relayURL, "on port", port);
});
```

- We'll keep this server in a file called `relay.js` in the same directory as our API server code. We can implement a Dockerfile (in `Dockerfile.relay`) for this server that's similar to the one we wrote for our original server:

```
FROM node:9
WORKDIR /usr/src/app
COPY package*.json ./
RUN npm install
COPY . .
ENV PORT=8001
```

```
EXPOSE ${PORT}
CMD [ "npm", "run", "relay" ]
```

- And we can build an image from this Dockerfile, specifying the option `-f Dockerfile.relay` to use a non-default Dockerfile:

```
docker build -t book-a-place-relay -f Dockerfile.relay .
```

- Next, we'll create a Docker network that we'll use to hook our relay server an API server together:

```
docker network create --driver bridge book-a-place-net
```

- Next, we'll start a Book-a-Place API server container. We'll do this similar to the way we did it before, except this time, instead of publishing port 8000 to the outside world, we'll instead put the container on the network we just created:

```
docker run -d                       \
     --name book-a-place-api        \
     --network book-a-place-net     \
     book-a-place-api
```

- Because we didn't publish port 8000, our API server is inaccessible to the outside world. For example, trying to make a request using cURL from our host machine would fail:

```
$ curl "http://loc:8000/lodgings"
curl: (7) Failed to connect to localhost port 8000:
Connection refused
```

- Now, we can start our relay server container. We'll need to make sure to put this container on the same network as the API server. We'll also need to specify the environment variables `RELAY_HOST` and `RELAY_PORT` to tell the relay server where the API server lives. This time, we'll publish port 8001:

```
docker run -d                             \
     --name book-a-place-relay            \
     --network book-a-place-net           \
     -e "RELAY_HOST=book-a-place-api"     \
     -e "RELAY_PORT=8000"                 \
```

```
    -p 8001:8001                          \
    book-a-place-relay
```

- Now, if we make a request to the *relay* server, we should get the API server's response:

```
$ curl "http://localhost:8001"
{"lodgings":...}
```

- We've successfully created a multi-container application here. Specifically, we have two different services running in separate containers, communicating with each other to provide a single application interface.

- Of course, this application itself is just a toy for demonstration purposes. However, you can imagine using a separate container to run our API server, one to run a MongoDB database, one to run an ElasticSearch index, etc., all being connected together via a Docker network. None of these services themselves would even know it was running inside a Docker container. Each would simply communicate over a network as it normally would. This is the beauty of Docker!