As a web developer, I’m sure you’ve built a server or two.[[1]](#footnote-1) You probably spun up a Linux box and installed some packages like Apache, MySQL, or PHP. Maybe you pulled some of your code from GitHub, threw a database together, and edited some config files to bend the server to your liking.

But what happens if your server becomes corrupted? Or if your backups have been failing without you noticing? Or if your site goes viral and traffic starts outpacing what your infrastructure can handle?

If you’re new to server administration, any of those events could seem daunting. If one occurred, you would have a bigger problem than manually rebuilding your server. You also have to hope your documentation is thorough enough to cover every modification you made to your original system. And you do have documentation, right?

Simply put, a disaster of any magnitude would make for a frustrating day or a late night. Disasters do happen and it’s prudent to be prepared for them.

Let me introduce you to Docker, which at it’s base is a container management system.

“But Henry, what in the world are containers and why should I care about them?”

Great question. Containers, in the context of software development, are best defined as segmented and sequestered user-space instances existing on top of a kernel.

“Yeah… can you make that simpler?”

Sure can. The easiest way to explain containers is to compare them to virtual machines (VM’s). One method to run three different apps on one server is to run three different VM’s, which consist of an operating system, the app, and whatever dependencies that app needs in order to run. A container, in the context of this example, is a bundle of the bare minimum of code and dependencies it takes to run an application. Docker functions as the guest operating system for any number or type of containers. The obvious benefit is that this is a lot more resource efficient than running several VM’s.

{ Fig 1: <https://images.idgesg.net/images/article/2017/06/virtualmachines-vs-containers-100727624-large.jpg> }

“Word. This is starting to make sense. Paint me a word picture."

You got it. Let’s dive in on how using Docker (and containers in general) are going to take your game to another level. The benefits to using Docker are fiftyfold, but for the sake of brevity I want to key in on the five I believe to be the most important.

1. SPEED:  
     
    The main key to wrap your head around is that Docker allows you to programmatically build out your infrastructure. That means instead of having to manually build another server every time you need a new one, you can just spin one up based on a predefined image of what you need the end result to be. It’s a little time intensive to build that image at the beginning of a new project, but every time you need to stand up a duplicate environment, it takes minutes instead of hours.
2. EASE OF USE:  
     
    Beyond speed, the ability to quickly spin up instances of infrastructure you need at any given moment can be a great boost to your organization. Let’s say the app you’re building hits the front page of Reddit and your traffic goes out of control. Or, alternatively, you get hacked and have your site knocked offline. Either way, you need to get back online, and fast. Your business very well might depend on it. At least nine out of ten times, starting a container is faster than building a fresh server.
3. SECURITY:  
     
    A basic tenet of containers is that they are essentially walled gardens. An initial benefit is that you can bundle apps with their respective dependencies, so that if Container A needs PHP 5 and Container B needs PHP 7, that can be handled without having to worry about dependency clashes. The two containers may run side by side, but they won’t get in each other’s way.  
     
    Additionally, that walled garden approach means you can define exactly how you want, if you want, data flowing in and out of a container. You’ll see in the demo that we explicitly tell our container to “EXPOSE 80”. That is saying it’s okay to open up port 80 in the container to the broader system we’re running it on, so that Apache can do it’s job and host content over that port. When you start building more complicated systems with Docker, you can use that kind of security to link certain containers running applications up to other specific containers holding databases or running some specific micro-services.
4. PORTABILITY:  
     
    Since containers (in a simplistic view) are meant to be an app packaged with the bare minimum of resources it needs to run correctly, you can run that container on anything that can run Docker. Whether it be on Red Hat Enterprise Linux, your MacBook Pro, or Windows Server 2016, that container will spin up and function exactly as it should.  
     
    An added benefit here is that you can develop in an environment that will eventually be your production environment. Your containers will be based on an image, which is a compiled list of instructions on how to build a specific environment. You can use that image to spin a container up on your laptop and build your app in it. Then when you’re ready to launch, you can use that same image to deploy a container on a public server and be completely confident that it’ll run the exact same there as it did locally.
5. VERSION CONTROLLABILITY:  
     
    And lastly, since Docker lets you build out your infrastructure as code, your infrastructure can be entered into version control systems. One more time for the people in the back, YOUR INFRASTRUCTURE CAN NOW BE VERSION CONTROLLED. If you make a change and something breaks, just roll it back. If your data center somehow burns to the ground, you can pull your images and code from your remote repositories. Future you is going to thank present you for making their life that much easier.

<!— START DEMO —>

So, with all that said, let’s build our first container. I’m assuming that I’ve convinced you of Docker’s usefulness enough that you’ve already downloaded and installed it on your system. If you haven’t yet, please go do that now. (<https://store.docker.com/search?type=edition&offering=community>)

Just for haha’s, let’s say I need a basic site on a CentOS server for a demo. There are two ways I can go about that: I can spend the twenty minutes it can take to create a new server from scratch, or I can use Docker to build one image that can then instantiate innumerable identical containers to run my site.

1. First, make a new project directory and create file named aptly “Dockerfile”, which is a list of instructions of how to build a system on top of a base image. Base images can be based on PHP, MySQL, or Ubuntu, and are generally provided by the maintainers of the source product. That means they’re built by the creators of the software to work well with Docker.  
      
    Then we need instructions on how to build on top of that base. There are commands we can use, like: RUN, ADD, and CMD. They run shell level commands, map files from the host to a container, and specify commands to run on boot, respectfully. Basically, we take the steps of configuring a server and translate them into tasks Docker can use to automate that process.  
     
    It’s similar to how Git works behind the scenes, in that each commit (or instruction) is a set of changes to be made to the last commit, not the entire codebase in that moment.  
     
    For the sake of simplicity, we’re going to be building a very basic image. As such, our Dockerfile will be as follows:  
     
    { Fig #2 }  
     
    A Dockerfile reads surprisingly like English. All it’s saying is that we want to use CentOS 7 as a base, install Apache, expose port 80 to map to a different host port, and start up Apache on boot.
2. Once the Dockerfile is saved, we’ll run a “build” command in the terminal. This tells Docker (the utility) to use our Dockerfile (the list of instructions) to build a system.  
     
    First, we need to open our terminal and traverse into the directory we’re using for our Dockerfile. For me, that directory is “~/Documents/LTP/DockerArticle/DockerDemo/“.  
     
    { Fig #3 }  
     
    Then, we need to tell the Docker utility to build the image being described by the Dockerfile.  
     
    { Fig #4 }  
     
    The [--rm] denotes that we want to automatically delete the intermediate containers that are generated during the build process, the [-t quinncuatro/DockerDemo] is tagging our build with a name so we can more easily refer to it later, and the [.] at the end denotes that we want to run this command from the present working directory.  
     
    This command will log out every part of the build process to the console. It may take a few minutes; that is totally normal. Let it sit for a minute, it will tell you when it’s finished.
3. Once Docker pulls down the base we chose to use (CentOS 7), installs Apache, and completes all other instructions, we’re given a finished image that is named based on the tag in the build command. You can see it (and the CentOS 7 base image) by listing the Docker images on your system.  
     
    { Fig #5 }
4. Now that the image is built, we can use it to spawn a running container. We’re going to change directory to the project we want running on this first container and then run a “run” command.[[2]](#footnote-2)  
     
    { Fig #6 }  
     
    This command is telling Docker to run a container based on the image we just built, but it has a few other parameters. The [-d] is telling Docker to run the container as a daemon (in the background), the [-p 80] is to map port 80 on the container to one on the host so a browser can reach it, the [--name Project1] is again a way to name the resource, and the [--mount type=bind,source=“$(pwd)”/app/,target=/var/www/html/] is telling docker to map our local “./app/“ directory to “/var/www/html/“ on the container so that Apache can host it for us.
5. Now that the container is running with our code loaded into it, we can go see it in a browser. First, we need to figure out what local port the container’s exposed port got bound to.  
     
    { Fig #7 }  
     
    We can see that the running container named “Project1” has mapped our local port 32788 to port 80 on the container. If we open a browser and head to “localhost:32788”, we can see our code running live.  
     
    { Fig #8 }
6. That’s really all there is to it. We could keep moving to different directories that have basic HTML/CSS/JS applications in a subdirectory called “app/“ and keep spawning containers. They’ll each spin up, map that code to “/var/www/html/“, and assign a port on “localhost” to each container.  
     
    And we can actually see verifiable that the container is pulling in our code by running a Docker command to get shell access to it. It’ll drop us right into the root of that CentOS system as the root user. If we look in the web directory, we can see that it pulled our files in and uses Apache to host them the same way we would on a traditional VPS.  
     
    { Fig #9 }
7. What demo would be complete without some file cleanup?  
     
    If we keep spawning containers without doing any sort of maintenance, we'll eventually going to hit the limits of our hardware. That’s not ideal. Let’s shut down the container we spun up, delete, and remove the image it’s based on.  
     
    Containers are ethereal. Once one is stopped with the “stop” command, we could spin it right back up, and it would be identical to the first time we started it. In order to get rid of a stopped container, we’d run an “rm” command. In order to get rid of the image it was based on, we would run an “rmi” command.  
     
    { Fig #10 }

<!— End Demo —>

Docker isn’t as insurmountable a technology as it seems. It can be tricky to wrap your head around, but once you do, it’s pretty easy to see the benefits it can provide. Not only does it let you develop locally in the same environment you’ll use for production, but it lets you spin your infrastructure up and down to suit your needs: whether it be scaling to meet viral demand or recovering from a disaster incident.

Further, this is just a very small taste of what Docker can do. Your builds can be as simple or as complicated as you need. For instance, I have one build that’s a group of three containers: one running a Node application, with another container running PostgreSQL with data persistence through the help of a data volume, and the third acting as a query cache.

As a developer, Docker is an incredibly powerful tool to have in your arsenal, even if it’s meant to solve a problem you haven’t had yet. You’ll eventually find yourself in a situation where Docker can save your bacon, and when you do, you’ll be ready for it. The documentation is incredible and it sounds corny, but the only real limit is your imagination. Now go forth and start building your infrastructure as code.

<!— ADDITIONAL RESOURCES —>

Article & Corresponding Code:

<https://github.com/Quinncuatro/TheFutureIsDocker.git>

Docker CE (Community Edition) Installation:  
<https://store.docker.com/search?type=edition&offering=community>

Getting Started Guides:

Mac - <https://docs.docker.com/docker-for-mac>

Windows - <https://docs.docker.com/docker-for-windows>

Docker Labs:

<https://github.com/docker/labs>

Dockerfile Documentation:

<https://docs.docker.com/engine/reference/builder/>

{{ UPDATE THE README }}

{{ GET ALL THE SCREEN GRABS }}

1. If you haven’t built a server and are interested in learning how, keep an eye out for my upcoming LearnToProgram.TV course titled “Intro to Server Administration”. [↑](#footnote-ref-1)
2. Here, I’m using a sample project from Dave Moran’s upcoming LearnToProgram.TV course titled “Something Something Sass”. [↑](#footnote-ref-2)