Docker for front end development

I've dabbled in working with Docker ever since it first became available for the Mac as Boot2Docker but I never actually did anything with it. Not too long ago I started working with the idea of creating a Docker app to store all the tools I use when writng or creating a front end application.

In building this container I will describe the basics of building a Dockerfile and using it to interact with your local filesystem. This will also go in a Github repo in case I need to duplicate the project at a later date.

Quick Docker overview

The idea of a container is that, like a Virtual Machine, it allows for full encapsulation of an application and its dependencies. For example if we're building a Node.js application with a MongoDB backend you can configure the container to download and install Node and MongoDB, perform any needed configuration and be ready to begin development. Furthermore you should get the same results any time that you build an image from a Dockerfile.

Installing Docker

Linux

There is no graphical installer for Linux like the ones we'll discuss later for Mac and Windows, you have two options: you can use the installer script our you can download a binary.

The install script is a shell script that will install Docker and associated dependencies. The process below uses Curl to download the script for your inspection and then run the script. We could automate the download and run into a single command but it's always a good idea to check the script matches the original source before running it.

Make sure to verify the contents of the script you downloaded matches the contents of install.sh located at https://github.com/Docker/Docker-install before executing.

Curl

If using Curl, run the following commands to install Docker.

```
curl -fsSL get.Docker.com -o get-Docker.sh
sh get-Docker.sh
```

If you want to download and install a test version of Docker use this command instead.

```
curl -fsSL test.Docker.com -o test-Docker.sh
sh test-Docker.sh
```

Wget

If you use Wget this is the command to download the production install script.

```
wget -qO- https://get.Docker.com > get-Docker.sh
sh get-Docker.sh
```

And this is the command to download the test/development version.

```
wget -qO- https://test.Docker.com > test-Docker.sh
sh test-Docker.sh
```

Downloading a binary

If you don't want to use the installer, or would like to use a different version of Docker than the one in the installer, you can download a binary from the Docker store. The downside to this approach is that you will have to manually install

updates.

Windows and Mac

Installing for Windows and Mac is as simple as downloading a binary package from the Docker store. Unlike the installer for Linux, the installers for Mac and Windows contain everything necessary to run Docker, build images and use other Docker tools.

Verifying successful installation

After installation start Docker and then, from a terminal, type `Docker version`. The result should be something similar to the block below.

```
$ Docker version
Client:
 Version:      17.06.1-ce
 API version:  1.30
 Go version:   go1.8.3
 Git commit:   874a737
 Built:        Thu Aug 17 22:53:38 2017
 OS/Arch:      darwin/amd64

Server:
 Version:      17.06.1-ce
 API version:  1.30 (minimum version 1.12)
 Go version:   go1.8.3
 Git commit:   874a737
 Built:        Thu Aug 17 22:54:55 2017
 OS/Arch:      linux/amd64
 Experimental: true
```

You are good to go.

Building the Dockerfile

Rather than create an empty image for a project and start from scratch every time

we can create a Dockerfile and use it as the basis for our our projects.

A Docker image is built up from a series of layers. Each layer represents an instruction in the image's Dockerfile. Each layer except the very last one is read-only.

Each layer is only a set of differences from the layer before it. The layers are stacked on top of each other. When you create a new container, you add a new writable layer on top of the underlying layers. This layer is often called the "container layer". All changes made to the running container, such as writing new files, modifying existing files, and deleting files, are written to this thin writable container layer.

The major difference between a container and an image is the top writable layer. All writes to the container that add new or modify existing data are stored in this writable layer. When the container is deleted, the writable layer is also deleted. The underlying image remains unchanged.

Because each container has its own writable container layer, and all changes are stored in this container layer, multiple containers can share access to the same underlying image and yet have their own data state.

In this case we want the container to do the following:

1. Install an image from Bitnami that already has Node installed
2. Install build tools, Bash, Curl, Git and Ruby
3. Force the container to use Bash
4. Install SASS and SCSS Lint using Ruby Gems
5. Install node-gyp and node-pre-gyp
6. Install a set of packages globally using Npm
7. Make a directory to store our code (named `app/code`) and set its (Unix) permissions
8. Copy a default `package.json` and `gulpfile.js to our work area
9. Install the packages specified in `package.json`
10. Expose port 3000 from our container to the outside world
11. Provide the user a bash shell to work from

To make the process easier to work through I've broken the Dockerfile into smaller chunks that roughly correspond to the items in the list. This will make the entire process easier to understand and reason through.

Install the image

The first step in any Dockerfile is to specify the base image we want to work from. In this case I've chosen to use the latest Bitnami image for Node rather than using a bare operating system. I loose the ability to install multiple versions of Node with NVM but I'd rather have one working version in the image than none at all so I'm ok with the compromise, for now.

```
FROM bitnami/node:8.4.0-r1
```

Install tools and utilities

Next we use the `apt-get` package manager to install tool and utilities for our container. These utilities include: `build-essentials` to handle compiling and linking in case the node packages have no binary version for Linux, `curl` to download files, the `bash` shell, `git` VCS, Ruby and Rubygems to handle two special cases.

There is a second block of libraries and applications that I'm installing to install other libraries and gems. `libffi6`, `libffi-dev` and `ruby-dev` are needed to install a Gem dependency.

```
# Fetch and install system tools
RUN apt-get update && apt-get -y -q --no-install-recommends install \
    build-essential \
    curl \
    bash \
    git \
    libffi6 \
    libffi-dev \
    python \
    python-dev \
    ruby \
    ruby-dev \
    rubygems
```

Force the container to use Bash

I prefeer to work with Bash and don't care about the standard shell that comes with Debian because it will not let me do completions, so I've added a layer to remove the default shell (`/bin/sh`) and create a symbolic link from Bash to make it the default font.

Now calling either `bin/bash` or `bin/sh` will run Bash. Problem solved, I think.

```
RUN rm /bin/sh && ln -s /bin/bash /bin/sh
```

Install SASS and SCSS Lint

Now that we've installed Ruby and Rubygems we can install the Ruby packages (gems) that we need for some of our Gulp tasks later on.

I know that there is a version of SASS that will work directly with Node and is written in C. I've used `node-sass` and the associated `libsass` but, despite what the authors say about functional parity between Ruby SASS and libsass some aspects of my tasks did not work so, rather than go through the hassle of debugging, I've chosen to stay with the Ruby version.

```
RUN gem install \
    sass \
    scss_lint
```

Make a directory and set its (Unix) permissions

Before we can install things we need to create a directory where to place our files. We use standard Unix commands to create a directory (`mkdir`) and set its permissions (`chmod`). Because this will be local to our image I've felt it was OK to make it publicly accessible (everyone can read, write and execute any file in the directory).

If you want to learn more about permissios or check for values that will make these permissions more restrictive, check Unix - File Permission / Access Modes

```
# Create the development directory
RUN mkdir /app/code/ && chmod 777 /app/code/
```

Copy defaults to work area

I hate reinventing the wheel so rather than create a new gulpfile and install packages manually for every project, I've copied my default `gulpfile.js` and `package.json` files to the root of my Docker development directory and use a layer to copy them into the container.

```
# If this works it should copy the package.json and gulpfile.js
# to the code directory
COPY package.json gulpfile.js /app/code/
```

Node related tasks

I'm usig four layers to complete all the Node associated tasks for this container. Since I changed the base image I no longer need to install NVM or Node (but I no longer have the capability to test code with different versions of Node) until I figure out how to install NVM at build time I'll consider this a fair trade.

The first two layers make sure that the tree benneath `/opt/bitnami/node/lib/` is publicly writeable to avoid possible errors.

We install packages for Node in 3 stages. The first stage will install `node-gyp` and `node-pre-gyp` to install binary packages. I have at least two of those packages in the list I install from `package.json` so I better do it now.

The next stage installs global packages. These will give me access to tools like Assemble, ESLint, Gulp and Netlify without having to create crazy npm script commands to make them work.

The task contiues by installing a small set of packages globally so we get command line tools for the following programs to work with:

- NPM (later version than what's installed with Node)
- Assemble
- ESLint

- Gulp
- Netlify
- NPM

Polymer and Firebase were part of the list but they triggered errors in NPM when working inside Docker so I removed them. If I need them for a particular project I can always install them from the shell.

After copying `package.json` I can install the packages listed there by running npm install. One thing to do periodically is update versions and make sure they still work, other than that the versions already there are known to work. This is the longest stage and the one where we could potentially find the most issues with, especially with dependencies that need to be compiled.

```
# Make the tree under /opt/bitnami/node/lib/ publically writeable
RUN /bin/bash -c "chmod -R 777 /opt/bitnami/node/lib/"
RUN /bin/bash -c "chmod -R 777 /opt/bitnami/node/lib/node_modules/"

# Install Gyp related tools for Node binary packages
RUN npm install -g \
    node-pre-gyp \
    node-gyp

# Install global packages
RUN npm install -g \
    assemble \
    eslint \
    gulp-cli \
    netlify-cli \
    npm

# Install packages from package.json
RUN npm install
```

Expose port from container to outside world

This layer uses EXPOSE to tell the world what port (or ports) the container will listen on. When we initialize the container we'll use this port to connect the container to our local development environment. Because the development server

I set up in Gulp uses port 3000 we'll use the same port.

```
# Expose default gulp port
EXPOSE 3000
```

Provide a bash shell for user to work from

Because we are providing interactive tools I want to make sure that the user of the container has a way to interact with the provided tools. We set up a WORKDIR pointing to our code directory and then use CMD to set up our default shell.

```
# Run with bash
WORKDIR /app/code/
CMD ["/bin/bash"]
```

Now to test.

Building the Docker image, creating a Github repo and pushing image to Docker Registry

Now that we know what we want to have in our image we can actually build it. The command is simple. What we want to do is change to the directory where you put your Dockerfile and run the following command"

```
Docker build -t front-end-dev -f Dockerfile.Docker .
```

The flags I used are as follows:

- **-t front-end-dev** assigns a name to the image.
- **-f Dockerfile.Docker** specified the location of the Docker file we'll use to build the image.

The period at the end (`.`) indicates the location where to build the image.

By default Docker caches the layers used to create an image. During development, where there may be many changes, you may consider disabling caching using **–no-cache=true**

We'll also create a Github repository to store the files. Right now there are only 3 but the number may increase as the project becomes more and more complex. Follow standard procedure:

1. Create the repository on Github
2. In your local directory initialize an empty Git repo (`git init`)
3. Add the project files (`git add .`) and commit them to the repo (`git commit -am "message here"`)
4. Push the files to the repository for the first time (`git push -u origin master`)

```
git init
git add .
git commit -am "initial message here"
git push -u origin master
```

The final stage is to push your image to the Docker Hub a central repository of Docker related images. This will also simplify building the container later on.

To make this easier I rebuilt the image with a namespace (my username in Docker Hub) and a tag name in addition to the image name, like so:

```
Docker build -t elrond25/front-end-dev:1.0 -f Dockerfile.Docker .
```

I logged into Docker Hub and created a project called `front-end-dev`.

Next, I logged in to the Docker hub from the command line:

```
Docker login -u elrond25 -p <password>
```

and finally I push the image to the registry

```
Docker push elrond25/front-end-dev:1.0
```

The public URL for the image is https://hub.Docker.com/r/elrond25/Docker-front-end/

Further ideas: Continous builds

Rather than rebuild and upload the image every time I make changes I've chosen to use Docker Hub's ability to build the image on the Hub whenever an associated Git (Github or Bitbucket) changes.

Configuring automated builds is a two step process. First you must link yourt Github and Docker Hub accounts:

1. Log into Docker Hub
2. Navigate to Profile > Settings > Linked Accounts & Services
3. Click the service you want to link (Github or Bit Bucket)
    1. The system prompts you to choose between Public and Private and Limited Access. The Public and Private connection type is required if you want to use the Automated Builds.
    2. Press Select under Public and Private connection type.
4. The system prompts you to enter your service credentials (Bitbucket or GitHub) to login

After you grant access to your code repository, the system returns you to Docker Hub and the link is complete.

The second step is to create the automated build

1. Select Create > Create Automated Build from Docker Hub
2. The system prompts you with a list of User/Organizations and code repositories
3. Select from the User/Organizations
    1. Optionally, type to filter the repository list
4. Pick the project to build
5. The system displays the Create Automated Build dialog
    1. The dialog assumes some defaults which you can customize. By default, Docker builds images for each branch in your repository. It assumes the Dockerfile lives at the root of your source. When it builds an image, Docker tags it with the branch name
6. Customize the automated build by pressing the Click here to customize this behavior link
7. Specify which code branches or tags to build from. You can add new

configurations by clicking the + (plus sign). The dialog accepts regular expressions.

8. Click Create.

Create container for new project

We now have a working image we have pushed into the Docker Hub so other people can share it. Now all we have left is to actually build a container with the image. I'm doing this last because I want to leverage the Hub to download a smaller image than I would if it was used locally.

```
Docker  run  -v `pwd`:`pwd` -w `pwd` -it --rm elrond25/front-end-dev:1.0
```

This time it's a single command, `Docker run` with the following parameters:

- The -v flag mounts the current working directory into the container
- The -w lets the command being executed inside the current working directory, by changing into the directory to the value returned by pwd
- –rm automatically remove the container when it exits
- -it instructs Docker to create an interactive bash shell in the container

So this combination executes the command using the container, but inside the current working directory. It will also provide a shell inside the container to run commands from.

Further ideas: Integrating MongoDB and other tools

So far we've only discussed building single use containers. One of the biggest attractions of Docker is that you can create applications using mulltiple containers using a separate tool called Docker Compose.

In this example we'll build a Wordpress application using two images: One for Wordpress 4.8.1 Running PHP 7.0 in Apache and one for MySQL 5.7. This wwill be similar to the example in the Docker samples site but tailored to run the latest versions of software.

Create a `stack.yaml` file and copy the content below to it:

```
version: '3.1'

services:

  wordpress:
    image: 4.8.1-php7.0-apache
    ports:
      - 8080:80
    environment:
      WORDPRESS_DB_PASSWORD: example

  mysql:
    image: mysql:latest
    environment:
      MYSQL_ROOT_PASSWORD: example
```

Then the command to bring this application up is:

```
Docker-compose -f stack.yaml up
```

This will automatically download the images (if needed) and configure them to give you a Wordpress installation ready for you to configure.

> This image does not provide any additional PHP extensions or other libraries, even if they are required by popular plugins. There are an infinite number of possible plugins, and they potentially require any extension PHP supports. Including every PHP extension that exists would dramatically increase the image size.

If you need additional PHP extensions, you'll need to create your own image FROM this one. documentation of the php image explains how to compile additional extensions. Additionally, the wordpress Dockerfile has an example of doing this. it looks like this:

```
# install the PHP extensions we need
```

```
RUN apt-get update && apt-get install -y libpng12-dev libjpeg-dev \
  && rm -rf /var/lib/apt/lists/* \
    && Docker-php-ext-configure gd --with-png-dir=/usr --with-jpeg-dir=/us
    && Docker-php-ext-install gd
RUN Docker-php-ext-install mysqli
```

Closing thoughts

The two examples in this post are fairly simmple as far as using Docker is concerned. They give us an idea of what you can do and starting point for further experimentation.

The next experiment is to create an image similar to `front-end-dev` but set up for Express and MongoDB development. Then we can compose that image with a MongoDB image from the Docker library and have a fully working development environment.

This also gives us the opportunity to play and experiment with applications we wouldn't normally use. I'm wondering if Apache Cassandra would work better than MongoDB. Now I can create stacks with both databases and test the code.

The sky is the limit... with moderation :-)

Links and Resources

- Docker Documentation
- Docker Reference Documentation
- Dockerfile Reference
- Front end development with Docker
- https://hub.Docker.com/r/elamoureux/frontend-js-dev/