# Technology Conversations

# Continuous Deployment: Implementation

*This article is part of the Continuous Integration, Delivery and Deployment series.*

The previous article described several Continuous Deployment strategies. In this one we will attempt to provide one possible solution for reliable, fast and automatic continuous deployment with ability to test new releases before they become available to general users. If something goes wrong we should be able to rollback back easily. On top of that, we'll try to accomplish zero-downtime. No matter how many times we deploy our applications, there should never be a single moment when they are not operational.

To summarize, our goals are:

- to deploy on every commit or as often as needed
- to be fast
- to be automated
- to be able to rollback
- to have zero-downtime

## Setting up the stage

Let's set-up the technological part of the story.

Application will be deployed as a Docker container. It is an open source platform that can be used to build, ship and run distributed applications.

While Docker can be deployed on any operating system, my preference is to use CoreOS. It is a Linux distribution that provides features needed to run modern architecture stacks. An advantage CoreOS has over others is that it is very light-wight. It has only few tools and they are just those that we need for continuous deployment. We'll use Vagrant to create a virtual machine with CoreOS.

Two specifically useful tools that come pre-installed on CoreOS are etcd (key-value store for shared configuration and service discovery) and systemd (a suite of system management daemons, libraries and utilities).

We'll use nginx as our reverse proxy server. Its templates will be maintained by confd that is designed to manage application configuration files using templates and data from etcd.

Finally, as an example application we'll deploy (many times) BDD Assistant. It can be used as a helper tool for BDD development and testing. The reason for including it is that we'll need a full-fledged application that can be used to demonstrate deployment strategy we're about to explore.

> *I'm looking for early adopters of the application. If you're interested, please contact me and I'll provide all the help you might need.*

## CoreOS

If you do not already have an instance of CoreOS up and running, continuous-deployment repository contains Vagrantfile that can be used to bring one up. Please clone that repo or download and unpack the ZIP file. To run the OS, please install Vagrant and run the following command from the directory with cloned (or unpacked) repository.


```
vagrant up
```

Once creation and startup of the VM is finished, we can enter the CoreOS using:

```
vagrant ssh
```

From now on you should be inside CoreOS.

## Docker

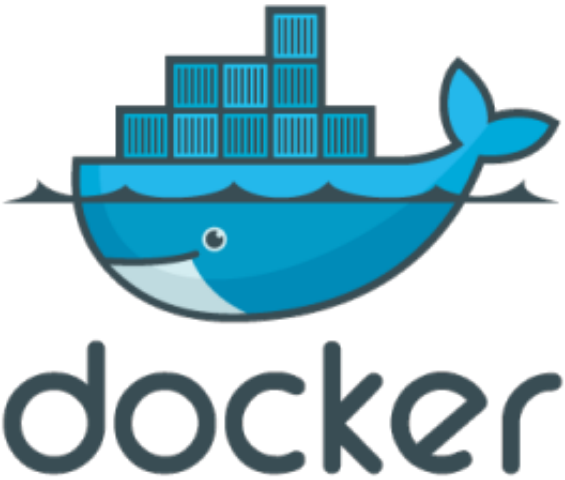
We'll use the BDD Assistant as an example simulation of Continuous Deployment. Container with the application is created on every commit made to the BDD Assistant repo. For now we'll run it directly with Docker. Further on we'll refine the deployment to be more resilient.

Once the command below is executed it will start downloading the container images. First run might take a while. Good news is that images are cached and later on it will update very fast when there is a new version and run in a matter of seconds.

```
# Run container technologyconversationsbdd and expose port 9000
docker run --name bdd_assistant -d -p 9000:9000 vfarcic/technologyconversationsl
```

It might take a while until all Docker images are downloaded for the first time. From there on,

starting and stopping the service is very fast. To see the result, open http://localhost:9000/ in your browser.



That was easy. With one command we downloaded fully operational application with AngularJS front-end, Play! web server, REST API, etc. The container itself is self-sufficient and immutable. New release would be a whole new container. There's nothing to configure (except the port application is running on) and nothing to update when new release is made. It simply works.

## etcd

Let's move onto the etcd.

```
etcd &
```

From now on, we can use it to store and retrieve information we need. As an example, we can store the port BDD Assistant is running. That way, any application that would need to be integrated with it, can retrieve the port and, for example, use it to invoke the application API.

```
# Set value for a give key
etcdctl set /bdd-assistant/port 9000
# Retrive stored value
etcdctl get /bdd-assistant/port
```

That was a very simple (and fast) way to store any key/value that we might need. It will come in handy very soon.

## nginx

At the moment, our application is running on port 9000. Instead of opening localhost:9000 (or whatever port it's running) it would be better if it would simply run on localhost. We can use nginx reverse proxy to accomplish that.

This time we won't call Docker directly but run it as a service through systemd.

```
# Create directories for configuration files
sudo mkdir -p /etc/nginx/{sites-enabled,certs-enabled}
# Create directories for logs
sudo mkdir -p /var/log/nginx
# Copy nginx service
```

```
sudo cp /vagrant/nginx.service /etc/systemd/system/nginx.service
# Enable nginx service
sudo systemctl enable /etc/systemd/system/nginx.service
```

nginx.service file tells systemd what to do when we want to start, stop or restart some service. In our case, the service is created using the Docker nginx container.

Let's start the nginx service (first time it might take a while to pull the Docker image).

```
# Start nginx service
sudo systemctl start nginx.service
# Check whether nginx is running as Docker container
docker ps
```

As you can see, nginx is running as a Docker container. Let's stop it.

```
# Stop nginx service
sudo systemctl stop nginx.service
# Check whether nginx is running as Docker container
docker ps
```

Now it disappeared from Docker processes. It's as easy as that. We can start and stop any Docker container in no time (assuming that images were already downloaded).

We'll need nginx up and running for the rest of the article so let's start it up again.

```
sudo systemctl start nginx.service
```

## confd

We need something to tell our nginx what port to redirect to when BDD Assistant is requested. We'll use confd for that. Let's set it up.

```
# Download confd
wget -O confd https://github.com/kelseyhightower/confd/releases/download/v0.6.3,
# Put it to the bin directory so that it is easily accessible
sudo cp confd /opt/bin/.
# Give it execution permissions
sudo chmod +x /opt/bin/confd
```

Next step is to configure confd to modify nginx routes and reload them every time we deploy our application.

```
# Create configuration and templates directories
sudo mkdir -p /etc/confd/{conf.d,templates}
# Copy configuration
sudo cp /vagrant/bdd_assistant.toml /etc/confd/conf.d/.
# Copy template
sudo cp /vagrant/bdd_assistant.conf.tmpl /etc/confd/templates/.
```

Both bdd_assistant.toml and bdd_assistant.conf.toml are in the repo you already downloaded.

Let's see how it works.

```
sudo confd -onetime -backend etcd -node 127.0.0.1:4001
cat /etc/nginx/sites-enabled/bdd_assistant.conf
wget localhost; cat index.html
```

We just updated nginx template to use the port previously set in etcd. Now you can open http://localhost:8000/ in your browser (Vagrant is set to expose default 80 as 8000). Even though the application is running on port 9000, we setup nginx to redirect requests from the default port 80 to the port 9000.

Let's stop and remove the BDD Assistant container. We'll create it again using all the tools we saw by now.

```
docker stop bdd_assistant
docker rm bdd_assistant
docker ps
```

## BDD Assistant Deployer

Now that you are familiar with the tools, it's time to tie them all together.

We will practice Blue Green Deployment. That means that we will have one release up and running (blue). When new release (green) is deployed, it will run in parallel. Once it's up and running, nginx will redirect all requests to it instead to the old one. Each consecutive release will follow the same process. Deploy over blue, redirect requests from green to blue, deploy over green, redirect requests from blue to green, etc. Rollbacks will be easy to do. We would just need to change the reverse proxy. There will be zero-down time since new release will be up and running before we start redirecting requests. Everything will be fully automated and very fast. With all that in place, we'll be able to deploy as often as we want (preferably on every commit to the repository).

```
sudo cp /vagrant/bdd_assistant.service /etc/systemd/system/bdd_assistant_blue@90
sudo cp /vagrant/bdd_assistant.service /etc/systemd/system/bdd_assistant_green@9
sudo systemctl enable /etc/systemd/system/bdd_assistant_blue@9001.service
sudo systemctl enable /etc/systemd/system/bdd_assistant_green@9002.service
# sudo systemctl daemon-reload
etcdctl set /bdd-assistant/instance none
sudo chmod 744 /vagrant/deploy_bdd_assistant.sh
sudo cp /vagrant/deploy_bdd_assistant.sh /opt/bin/.
```

We just created two BDD Assistant services: blue and green. Each of them will run on different ports (9001 and 9002) and store relevant information to etcd. deploy_bdd_assistant.sh is a simple script that starts the service, updates nginx template using conf and, finally, stops the old service. Both BDD Assistant service and deploy_bdd_assistant.sh are available in the repo you already downloaded.

Let's try it out.

```
sudo deploy_bdd_assistant.sh
```

New release will be deployed each time we run the script deploy_bdd_assistant.sh. We can confirm that by checking what value is stored in etcd, looking at Docker processes and, finally, running the

application in browser.

```
docker ps
etcdctl get /bdd-assistant/port
```

Docker process should change from running **blue** deployment on port **9001** to running **green** on port **9002** and the other way around. Port stored in etcd should be changing from 9001 to 9002 and vice verse. Whichever version is deployed, http://localhost:8000/ will always be working in your browser no matter whether we are in the process of deployment or already finished it.

Repeat the execution of the script deploy_bdd_assistant.sh as many times as you like. It should always deploy the latest new version.

For brevity of this article I excluded deployment verification. In “real world”, after new container is run and before reverse proxy is set to point to it, we should run all sorts of tests (functional, integration and stress) that would validate that changes to the code are correct.

## Continuous Delivery and Deployment

The process described above should be tied to your CI/CD server (Jenkins, Bamboo, GoCD, etc). One possible Continuous Delivery procedure would be:

1. Commit the code to VCS (GIT, SVN, etc)
2. Run all static analysis
3. Run all unit tests
4. Build Docker container
5. Deploy to the test environment
    1. Run the container with the new version
    2. Run automated functional, integration (i.e. BDD) and stress tests
    3. Perform manual tests
    4. Change the reverse proxy to point to the new container
6. Deploy to the production environment
    1. Run the container with the new version
    2. Run automated functional, integration (i.e. BDD) and stress tests
    3. Change the reverse proxy to point to the new container

Ideally, there should be no manual tests and in that case point 5 is not necessary. We would have Continuous Deployment that would automatically deploy every single commit that passed all tests to production. If manual verification is unavoidable, we have Continuous Delivery to test environments and software would be deployed to production on a click of a button inside the CI/CD server we’re using.

## Summary

No matter whether we choose continuous delivery or deployment, when our process is completely automated (from build through tests until deployment itself), we can spend time working on things that bring more value while letting scripts do the work for us. Time to market should decrease drastically since we can have features available to users as soon as code is committed to the repository. It's a very powerful and valuable concept.

In case of any trouble following the exercises, you can skip them and go directly to running the deploy_bdd_assistant.sh script. Just remove comments (#) from the Vagrantfile.

If VM is already up and running, destroy it.

```
vagrant destroy
```

Create new VM and run the deploy_bdd_assistant.sh script.

```
vagrant up
vagrant ssh
sudo deploy_bdd_assistant.sh
```

Hopefully you can see the value in Docker. It's a game changer when compared to more traditional ways of building and deploying software. New doors have been opened for us and we should step through them.

This article used CoreOS as operating system. However, in many cases companies prefer to stick to OS they already use (Red Hat, Ubuntu, etc). While they are not designed specifically to act as a host to Docker containers, similar principles as described above can apply. We just need to use slightly different approach. That is the topic of the next article. How to apply blue-green deployment with Docker containers to Ubuntu. As a cherry on top, we'll use Ansible as Configuration Management tool. We'll do Continuous Deployment implementation with Ansible and Docker.

This entry was posted in Continuous Integration, Delivery and Deployment and tagged BDD Assistant, CD, CI, confd, Continuous delivery, Continuous integration, Continuous Integration, Delivery and Deployment, CoreOS, Docker, etcd, systemd on December 8, 2014 [http://technologyconversations.com/2014/12/08/continuous-deployment-implementation/] by Viktor Farcic.

## 3 thoughts on "Continuous Deployment: Implementation"

Pingback: 1p – Continuous Deployment: Implementation – Offeryour.com Blog

jmasramon
December 8, 2014 at 7:41 pm

Great article Viktor. I will implement this in my new project for sure.

Mourn Grym
January 8, 2015 at 12:19 am

So, I like this article as a discussion of how Docker is ready for production deployments. As a functional wrapper around the stable Linux Containers implementation, I think it has a lot of advantages.

Having said that, I continue to see articles on continuous deployment or continuos integration with the magic “zero downtime”. However the example discussions are always application based. Particularly with online CMS sites, a major component that supports those sites is being overlooked (I suspect because there isn’t an easy answer) – the database.

We all know whether we use WordPress, Drupal, Pressflow, DNN, et al. CMS platforms OR any homegrown site with a database backend, that upgrades generally come with simple to very complex database updates (table additions ore removals, column changes, index updates – major schema related work particularly when done to databases with existing data). The problem is “how do I apply my new build with these dependent DB changes so that I have the ability to do continuous integration/deployment with zero downtime”.

The general answer is almost always flipping from one database to another (either a slave in a two node master/slave cluster or a node in a multi node cluster like Galera that was not part of the “live” rotation and had been removed to apply updates to). The challenge remaining is that last step of flipping over, or at the very least, rolling back without having to take things offline to restore last full backups (or backups taken before the build).

What I want is an honest discussion of that scenario. The fact is there is no easy answer for how to make major database changes in conjunction with a build and maintain this mythical “zero downtime” credo that has been sold to the IT executive upstream. If there is and we are just missing it, I would love to hear it discussed in detail.