12 Factor Applications

with Docker and Go

Tit Petric

Step by step guide for
12FA compliant Go apps.

12 Factor Applications with Docker and Go

A book filled with examples on how to use Docker and Go to create the ultimate 12 Factor applications

Tit Petric

This book is for sale at http://leanpub.com/12fa-docker-golang

This version was published on 2020-07-14

This is a Leanpub book. Leanpub empowers authors and publishers with the Lean Publishing process. Lean Publishing is the act of publishing an in-progress ebook using lightweight tools and many iterations to get reader feedback, pivot until you have the right book and build traction once you do.

Tweet This Book!

Please help Tit Petric by spreading the word about this book on Twitter!

The suggested hashtag for this book is #12fadocker.

Find out what other people are saying about the book by clicking on this link to search for this hashtag on Twitter:

#12fadocker

Contents

Introduction

About me

I'm one of those people with about two decades of programming experience under my belt. I've started optimizing code in the 90's, discovered PHP in the 2000's, and build several large-scale projects on to the day, all while discovering other programming language families like Java, Node.js and ultimately, Go.

I have built numerous APIs for my own content management products. Several products I've been involved with as a lead developer have been sold and are used in multiple countries. I've written a professional dedicated API framework which doubles as an software development kit for the Slovenian national TV and Radio station website, RTV Slovenia. I'm also the speaker at several local PHP user group events and conferences.

I am also the author of API Foundations in Go[1].

I write a blog which is available on scene-si.org[2], you should check it out, I tend to publish new articles about twice monthly.

Who is this book for?

This book is for everyone who writes applications for a living. I cover a wide area of subjects dedicated to development of software in general, trying to establish some good development practices, while at the same time referencing the 12 Factor App manifesto.

In the book, I will cover these subjects:

- I. Codebase - One codebase tracked in revision control, many deploys
- II. Dependencies - Explicitly declare and isolate dependencies
- III. Config - Store config in the environment
- IV. Backing services - Treat backing services as attached resources
- V. Build, release, run - Strictly separate build and run stages
- VI. Processes - Execute the app as one or more stateless processes
- VII. Port binding - Export services via port binding
- VIII. Concurrency - Scale out via the process model
- IX. Disposability - Maximize robustness with fast startup and graceful shutdown

[1] https://leanpub.com/api-foundations
[2] https://scene-si.org

- X. Dev/prod parity - Keep development, staging, and production as similar as possible
- XI. Logs - Treat logs as event streams
- XII. Admin processes - Run admin/management tasks as one-off processes

Covering these concepts should give you a good overview of the 12 Factor App designs and what problems they are trying to solve. The book tries to give you a set of best practices to follow and guide you through individual chapters shedding light and hands on examples of individual approaches.

How should I study it?

Through the book, I will present several examples on how to do common things when developing APIs. The examples are published on GitHub[3].

You should follow the examples in the book, or you can look at each chapter individually, just to cover the knowledge of that chapter. The examples are stand-alone, but generally build on work from previous chapters. Be sure to follow the Requirements section as you're working with the book.

[3] https://github.com/titpetric/books

Requirements

This is a book which demonstrates 12 factor application development on the use case of developing and deploying microservices. As such, there are a few prerequisites that need to be taken care of, when you'll be going through the examples in the book.

Linux and Docker

The examples of the book rely on a recent docker-engine installation on a Linux host.

Own hardware

The recommended configuration if you have your own hardware is:

- 2 CPU core,
- 2GB ram,
- 128GB disk (SSD)

The minimal configuration known to mostly work is about half that, but you might find yourself in a tight place as soon as your usage goes up. If you're just tying out docker, a simple virtual machine might be good enough for you, if you're not running Linux on your laptop already.

If you're running windows 10, you can enable the Hyper-V service, which allows you to install and run Linux in a virtual machine on your laptop/PC, I highly recommend this option.

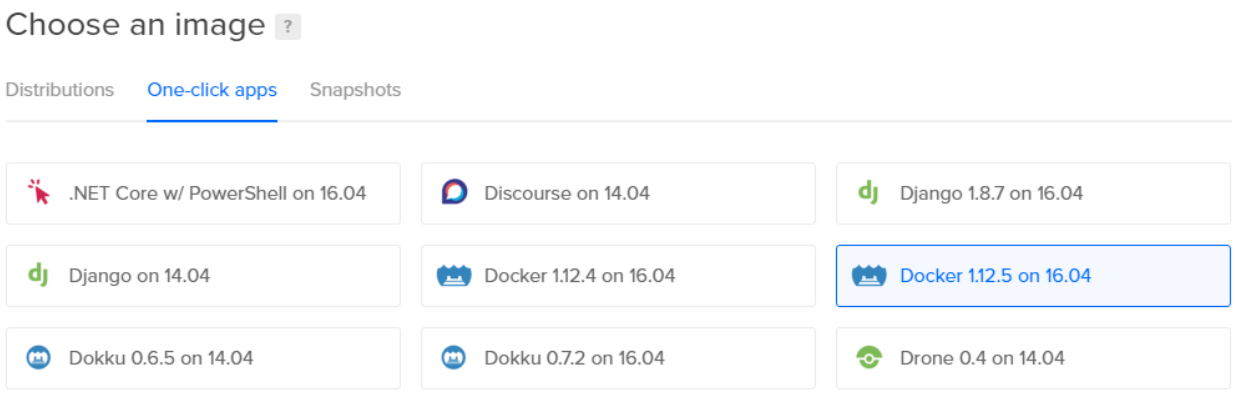
Please refer to the official docker installation instructions[4] on how to install a recent docker version.

Cloud quick-start

If having your own hardware is a bit of a buzzkill, welcome to the world of the cloud. You can literally set up your own virtual server on Digital Ocean within minutes. You can use this DigitalOcean referral link[5] to get a $10 credit, while also helping me take some zeros of my hosting bills.

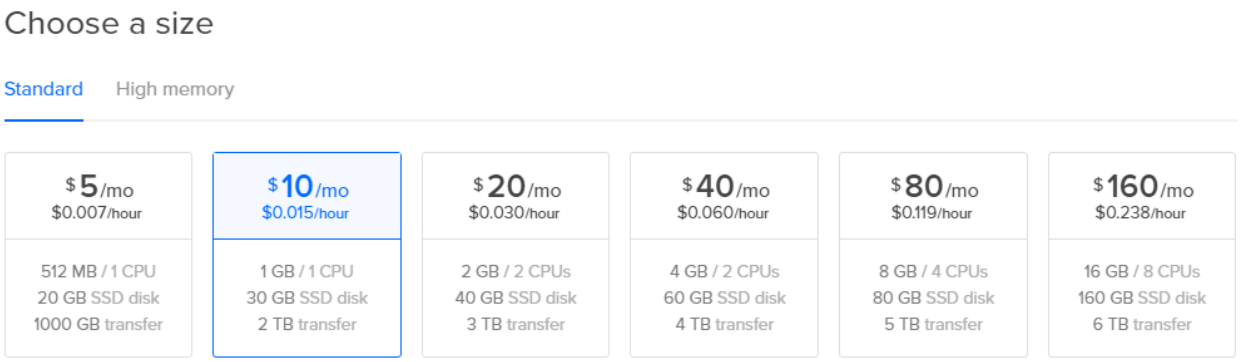
After signing up, creating a Linux instance with a running Docker engine is simple, and only takes a few clicks. There's this nice green button on the top header of the page, where it says "Create Droplet". Click it, and on the page it opens, navigate to "One-click apps" where you choose a "Docker" from the list.

[4] https://docs.docker.com/engine/installation/linux/
[5] https://m.do.co/c/021b61109d56



Choose Docker from "One-click apps"

Running docker can be disk-usage intensive. Some docker images may "weigh" up to or more than 1 GB. I would definitely advise choosing an instance with *at least* 30GB of disk space, which is a bargain for $10 a month, but you will have to keep an eye out for disk usage. It's been known to fill up.



Choose a reasonable disk size

Aside for some additional options on the page, like chosing a region where your droplet will be running in, there's only a big green "Create" button on the bottom of the page, which will set up everything you need.

Creating Digital Ocean droplets via API

Digital Ocean provides the CLI utility doctl[6]. It's a Go program that interfaces with their API to allow you to list running droplets, create new ones, shut them down and much more. If you don't need an instance 24/7, you can use the API to spin it up and shut it down based on your needs. It's good for having better quality development machines, which you delete after you're done for the day, saving about half of the costs.

Download and install `doctl` from the releases page and issue `doctl auth init` to authenticate against the Digital Ocean API. You can generate the token from the top of your dashboard.

[6] https://github.com/digitalocean/doctl

```
# doctl auth init
DigitalOcean access token: 24df63816084cb8270fbe03a7c98341a
Validating token: OK
```

After this, you're free to list images and create new droplets via the API.

```
# doctl compute image list | grep docker
23219707    Docker 17.03.0-ce on 14.04          docker
24232340    Docker 17.04.0-ce on 16.04          docker-16-04
```

We can use the `docker-16-04` image to quickly spin up a droplet. We will create `create.sh` and `destroy.sh`, which will check for existance of the droplet before it's created or destroyed.

Create a new droplet conditionally

```
#!/bin/bash
CHECK=$(doctl compute droplet list asx --format ID --no-header | wc -l)
if [ "$CHECK" == "0" ]; then
        echo "Creating ASX droplet"
        doctl compute droplet create asx -v \
                --image docker-16-04 \
                --size 2gb \
                --region ams3 \
                --ssh-keys $(./ssh-key.sh)
else
        echo "Droplet ASX already running"
fi
```

You should change the parameter to `--ssh-keys` with either your SSH key or a SSH key fingerprint, once you've added the SSH key to the Digital Ocean settings page under "Security". You can adjust the parameters based on the size of the droplet you would like and the region where you want it to run.

Destroying the droplet is just as simple as creating one:

Destroy a droplet if it exists

```
#!/bin/bash
CHECK=$(doctl compute droplet list asx --format ID --no-header | wc -l)
if [ "$CHECK" == "0" ]; then
        echo "Droplet ASX not started"
else
        echo "Deleting droplet asx"
        doctl compute droplet delete asx -f -v
fi
```

Of course, when your droplet spins up, you'll want to list it's public IP, or connect to it via SSH.

Listing the public ip for a droplet

```
doctl compute droplet list asx --format PublicIPv4 --no-header
```

Connecting to the IP is just as simple as wrapping everything in a `ssh $( + )`.

Connect to the droplet

```
ssh $(doctl compute droplet list asx --format PublicIPv4 --no-header)
```

As soon as you SSH into the instance, you can use any `docker run` command you like. At this point it's up to you to set up any git checkouts or somehow populate the running droplet.

All the scripts to create Digital Ocean instances programmatically are available in the book samples GitHub repository[7].

Additional requirements

Networking

When dealing with microservices in docker, it's a very common practice to enable communication between one service and another. For example, many microservices may and do communicate to each other. For this purpose we create a network called "party", on which the microservices from this book will run on.

[7] https://github.com/titpetric/books/tree/master/12fa-docker-golang/digitalocean

Create a custom bridge network

```
#!/bin/bash
#
# this creates a custom docker "bridge" network named "party"
#
docker network create -d bridge --subnet 172.25.0.0/24 party
```

When running microservices, it is enough to join the docker container to this network with the option `--net=party`. You may define multiple networks to isolate docker containers away from each other.

MySQL database

Some services will require a MySQL database to store data. We can quickly start up an instance in docker as well. We will name our instance "mysql" and will try to re-use it when ever an opportunity arises.

Quickstart: Run a MySQL instance

```
#!/bin/bash
NAME="mysql"
APP_DIR=$(dirname $(readlink -f $0))
mkdir -p ${APP_DIR}/data/$NAME
docker stop $NAME
docker rm $NAME

DOCKERFILE="titpetric/percona-xtrabackup"

docker stop $NAME
docker rm $NAME
docker run --restart=always \
    -h $NAME \
    --name $NAME \
    --net=party \
    -v ${APP_DIR}/data/$NAME/data:/var/lib/mysql \
    -e MYSQL_ALLOW_EMPTY_PASSWORD="yes" \
    -d $DOCKERFILE
```

When using the instance from other docker containers on the same network, you may use the following credentials to connect to it:

- hostname: `mysql` (the same as the container name),

- username: `root`,
- password: empty string

> Note: obviously having a MySQL instance with an empty root password in production is bad form. If you'd like to run MySQL with Docker in production, take a look at the default options in the percona image[a] on Docker Hub. There's a slightly better example of using MySQL in chapter 4 (backing services). Please consult that chapter for more examples.
>
> [a] https://hub.docker.com/_/percona/

I. Codebase - One codebase tracked in revision control, many deploys

Source code VCS (version control systems) are designed to store revisions of your source code. For each feature you add, they log a "commit" or version, which is a point in time snapshot of your changes. Commits can be recalled at a later time in case you need to roll-back your code for any reason, including getting a deleted section of source code restored at a later time.

Commits, as they are only partial snapshot on the changes on your code base, are an ideal way to enable collaborative work on software projects - if user 1 edits file 1 and user 2 edits file 2, there can be no conflict when merging commits between these two users. And when it comes to editing the same file - users work on local copies, and when they push their commits, a conflict might occur. Resolving the conflict may be as simple as editing a file, picking the lines which should stay and committing the changes.

I'd suggest you use `git` for revision control. You can use services like GitHub[8] or Bitbucket[9], or host your own with GitLab[10] or Gogs[11]. Make sure that you perform a full backup on occasion and have procedures in place which allow you to restore it from a backup. This should go without saying.

For company use, I'd recommend starting with a Bitbucket, which is free up to 5 users, and allows unlimited private repositories. If you want more than the 5 free users, the pricing is $10/month for 10 users, $20/month for 20, and so on.

If you're developing for/with community, I'd definitely recommend you to use GitHub instead. While pricing for private repos is higher, GitHub also provides a coherent issue tracker, lots of tooling to support your development, and an invaluable community. A team account with 5 users starts with $25/month.

Your application project or microservice should use one repository just for the sake of simplicity. You can use multiple repositories in a project, but it will increase complexity, usually not with good results. You can always structure your Go application in subpackages and keep everything neatly organized. If you are making several applications that use the same parts of code, this code should be separated into packages which may be included via `go get` or preferably a dependency manager (See chapter 2).

Let's to through setting up a self-hosted git service, where we can start work right away.

[8] https://github.com
[9] https://bitbucket.org
[10] https://gitlab.com
[11] https://gogs.io

Gogs

Gogs is a great option if you want a zero-cost self-hosted git solution. I'm going to walk you through the process of setting up Gogs. And since this is a book that relies heavily on docker, we'll use it to set up our Gogs instance.

Why Gogs in the first place?

As mentioned, there are a lot of services that are the standard today, most notably GitHub and Bitbucket. Bitbucket is a great option if you want a free plan that allows private repositories and managing permissions in a small team of up to 5 people, and they give you three extra people if you invite them to sign up. But, setting up your own git repositories gives you the benefit of avoiding things like rate limiting or issues with these services. If you're fanatic about keeping your code off the cloud, hosting it yourself is also a good option (but don't skip the section on backing it up). If your needs tend to grow (as they do), it's always an easy task to migrate from Gogs to GitLab, GitHub or Bitbucket at a later time, and I'm going to show you how to do that as well.

Installing and running Gogs

Gogs is written in Go. This means, they have a nice little set of binaries available for most common distributions and architectures, including Windows, if you like. But as we like the comfort of a process manager around Gogs, we will run it in docker. I've created a script to do the following for convenience:

Quick start: Run a Gogs instance

```
#!/bin/bash
NAME="gogs"
APP_DIR=$(dirname $(readlink -f $0))
mkdir -p ${APP_DIR}/data/$NAME
docker stop $NAME
docker rm $NAME
docker run -d --net=party --restart=always --name=$NAME -h $NAME -p 10022:22 -p 3000\
:3000 -v ${APP_DIR}/data/$NAME:/data gogs/gogs
```

The process manager part here is the `--restart` option, which instructs docker to restart the container should it exit for any reason. A possible reason being, that the server was rebooted for maintenance. After executing this script, Gogs is running in a docker container with the same name, exposing ports 10022 for ssh access, and port 3000 for http access.

When you open `http://yourserver:3000/` an install process will guide you through the installation. In the options, you may configure it by connecting to the MySQL instance we created in the

Introduction chapter, or you may use a `SQLite3` back-end, to store it's data locally without a database server.

In case you ran the `mysql` service from the Introduction chapter, you will need to create a `gogs` database, which will store the table schema and data for the Gogs application:

```
1 docker exec mysql mysqladmin create gogs
```

> Note: When it comes to production use, individual usernames should be created for individual microservices, only giving access to the database of the microservice. That way, a `gogs` user could only access the database with the same name. If a microservice needs data from another database, it should reasonably get this data from another microservice (Consult Chapter IV. Backing Services).

Logging in

Just in case if you didn't enter an administrative user in the installation section: The first user who will be registered in Gogs, will become the administrative user. There's no default user created.

Creating a repository

We can create a new repository by clicking the "+" next to "Repositories" (on the right side). As a test we can create a 'the12factors' repository. Gogs will give you nice instructions how to create a new repository or push an existing repository to it. For our purposes, we will create our own repository:

```
1 mkdir the12factors
2 cd the12factors
3 touch README.md
4 git init
5 git add README.md
6 git commit -m "first commit"
7 git remote add origin http://peer.lan:3000/titpetric/the12factors.git
8 git push -u origin master
```

The repository will be created and new changes available

> Note: every git push will ask for username if you use the http method. It's possible to save these credentials on the system, so it will not ask you for them every time. A better method is to use the `ssh` endpoint, where authentication is done with an ssh key.

Backing up Gogs

If you set up Gogs with an SQLite database as suggested, the backup of Gogs is very simple:

Quick start: Backup a full Gogs instance

```
#!/bin/bash
NAME="gogs"
docker stop $NAME
cd data && tar -zcvf gogs-backup-$(date +"%Y%m%d").tgz gogs/ && cd ..
docker start $NAME
```

The backup script stops the Gogs instance, so all the data is written to disk, and then creates a backup archive with the current data in the `data/` folder. This data can be archived, or you can use it for migration to another service.

> Note: be sure to back up your MySQL instance in case you used it to set up Gogs. The database contains all the schema and data you will need to restore it on another server. You may use the same principle for backing up mysql data as we're doing with Gogs. These backup files will be the raw backup files that MySQL stores on disk, so you will need to pay attention that you're restoring the data into a compatible MySQL version, the same version if possible.

Multiple deploys

When you're are using git for version control, there are a few accepted methodologies of how to set up your code so that you can have manageable deploys to several environments: development, staging, testing, production.

The simplest workflow to implement is just to have a master branch, and whatever lives there is ready for production. New development features should be done on other branches. Ideally, the master branch would be read-only, and feature branches should be merged with pull requests. Pull requests enable code reviews, commenting and discussion before accepting some change into production.

Ideally what would happen with multiple developers:

Fork, clone and branch the repository

Depending on your project, you might want to enforce forking a sensitive repository, which you don't want trashed by developers missteps. A fork is a literal copy of the complete contents of a git repository. This fork is usually created via the web interface on GitHub, GitLab, Bitbucket, Gogs and other git repository services. Each developer should have read-only access to the main source repositories.

If you don't enforce forks, you should enforce a read-only master branch. GitHub as well as Bitbucket allow you to do this, most likely GitLab and Gogs as well. Consider this something like a contract between your senior and junior developers. You should definitely enforce some limitations on the master branch that would prevent history rewrites and deletion, the logical conclusion is to have in read-only so that bad things could only happen on branches, which can effectively be deleted later.

Creating a branch on your cloned repository is as simple as running one command:

```
# git checkout -b new-feature
Switched to a new branch 'new-feature'
```

and if you want to use an existing branch, just skip the `-b` option. Inspect which branch you're working on by running `git branch`:

```
# git branch
  master
* new-feature
```

All the changes which you will now commit with git will be created in this branch. When using `git push` only the current branch will be pushed.

Squashing your changes

If your feature required several commits to finish, you might want to squash it into one commit, before you push the changes upstream. What this does is perform a local merge which will create a single commit instead of several which you choose. You can perform the squash by running:

```
# git rebase -i HEAD~2
```

Where the number 2 is the amount of commits you want to squash. You will be given a prompt with something similar to the following:

```
pick 255ac52 add title to readme
pick 3e9f63e add contents
```

And to actually squash the commits, you would change the trailing “pick” lines into “squash”:

```
pick 255ac52 add title to readme
squash 3e9f63e add contents
```

and then modify your commit message accordingly. The command will finish with something like this:

```
# git rebase -i HEAD~2
[detached HEAD cd3041c] add title to readme add contents
 Date: Mon Apr 17 08:16:00 2017 +0200
 1 file changed, 3 insertions(+)
Successfully rebased and updated refs/heads/new-feature.
```

We can inspect that the individual histories no longer exist:

```
# git log --oneline
cd3041c add title to readme add contents
525f2c9 import readme
```

Only the squashed commit is available on the branch now and you can push the changes to your branch.

Pushing changes

Pushing changes is as simple as issuing a `git push`. If you already pushed your commits, you will have to use a forced push to your branch:

```
# git push origin my-feature --force
```

Keep in mind, just like with the `master` branch, this is quite dangerous. If you're relying anywhere on `git pull`, this will break that functionality. When you rewrite history, it will break all the checkouts that already know about the commits which you're rewriting. For those, you will have to clone the repositories anew.

After you push your changes, a PR can be submitted to merge your changes to the master branch. These PRs are a good opportunity where you can introduce code reviews and automated testing. More about that in Chapter V.

Git fork/branch simplified

1. Create a fork
2. Clone the repository
3. Create a branch
4. Commit, squash and push changes
5. Submit a pull request

Other

I'm a proponent of feature branches, but there are other branching strategies available for you, depending on your product requirements. You may consider a *release branching* model, where work is done on branches which follow some naming convention, like semantic versioning.

> Note: More information about semantic versioning can be found on semver.org[a].
>
> [a] http://semver.org/

When your application and team get bigger, you should consider extending this practice with Git-Flow[12] or pick and choose a branching strategy that works for you.

[12] http://nvie.com/posts/a-successful-git-branching-model/

Structuring your code

As we will be using the Docker image `golang` for our examples, I will now explain how your project may be laid out, by explaining some things that go assumes about how you lay out your code. When you will use native vendoring, it means that you can continue to use `go run` and `go build`.

The GOPATH environment is set to `/go`. Your package layout should be something like this:

```
$GOPATH(/go)/src/app
  |__ main.go (Where your main code lives)
  |__ subpackage/
  |     |__ subpackage.go (You can create subpackages)
  |     \__ ...
  |__ ... (Other files or subpackages)
  \__ vendor/
        |__ github.com/namsral/flag
        |__ github.com/gorilla/mux
        \__ ...
```

Using subpackages is as simple as using `import "app/subpackage"`, where `app` is the folder name where your application is checked out. It's also possible to use a fully qualified domain name for the packages which are imported. In this case, the packages are installable in other apps, via commands like `go get` or vendoring tools like `godep` or `gvt`. More about this later.

Importing subpackages to your application

```
package main

import "app/apiservice"

func main() {
  apiservice.HelloWorld()
}
```

With our docker example, we use `app` for our project name. Obviously, this is up to you - but as you're creating a docker container for each run, we put whatever folder you have into `/go/src/app`.

Running your app with go run and docker

```
#!/bin/bash
docker run --rm -it -v `pwd`:/go/src/app -w /go/src/app golang go run main.go
```

When structuring the code like this, there are a few things I'd like to point out.

Relative imports

Don't use relative imports. In your main.go instead of `import "app/subpackage"` you could use `import "./subpackage"`, but it would break functionality. Relative imports at the time of writing don't work well with `vendoring`, which is a required step when building 12 factor apps (define and isolate dependencies).

Using subpackages externally

If we have multiple subpackages, let's say `subpackage1` and `subpackage2`, we shouldn't import these from subpackages with the same pattern. It will work for your application, but it will not work for an application which would use your subpackages with an import from a VCS.

Example of a problematic subpackage

```
1  package subpackage1
2
3  import "fmt"
4  import "app/subpackage2"
5
6  func Hello() {
7    return subpackage2.Hello();
8  }
```

We would use this subpackage from another application with something similar to this:

```
1  package main
2
3  import "example.github.com/username/your-app/subpackage1"
```

When you're doing this, you will have a problem that your new application knows nothing about "subpackage2", which is referenced in "subpackage1". There are only a few ways to fix this:

1. Use fully-qualified notation when importing other packages from subpackages,
2. Your packages should be self-contained (no importing subpackages from parent project)

There are various ways to achieve independence from other subpackages, while at the same time having a decoupled relationship with them. The most common way might be to use some sort of dependency injection. With this approach, your submodules would not be bound to other submodules (coupling), but would define an interface which can be satisfied with another submodule (loose coupling).

Improving deployments

When you're starting out with deployments, it's a very good idea to enforce some automation from the very beginning. Using a CI system is a must-have.

People use various continuous integration tools like Jenkins[13] or Travis[14], or cool hosted services like Buildkite[15] or Codeship[16]. It's not very hard to make tests when you're just starting to write your application, it's much harder when your application is already operational. Add tests in your CI workflow, so you don't publish any project which has tests that are failing.

The simplest way to test your Go application is running `go test`, but this is only one part of the testing which you need to pay attention to. If you're building a Docker image, it's a good idea that you set up a testing procedure for it as well.

Make sure that you can perform your build in one step. Usually it's enough to include a "build.sh" at the root of your project and then use any CI suite to run it. Depending on the size, this may invoke tools like `ant` or `make`, but when we're talking about microservices, we're talking about small applications that have a single responsibility, and not a big application that handles everything from data storage, session storage, templating, separate functionality for individual business areas. Ideally, each of these would be represented as a single microservice, so it can be replaced or optimized to take advantage of new underlying technology used.

Your testing procedures *must* be a condition of deploying an application. If the tests don't pass, the person who triggers the deploy has to be notified. It doesn't really matter who fixes the application or tests or when the fix is made, but until it's done - no deployments should be possible. This doesn't have to be a critical scenario in the sense of urgency to fix it right away, but can be done within weekly sprints.

That being said, the goal is to have individual microservices be deployable at any given moment, and a gold rule of thumb for sprints is that you really shouldn't deploy them on fridays, or on days where critical members of your team leave for vacation. There's a thing to be said on catching deadlines, but also planning on time to fix any issues which might occur after deployments and were not caught by tests.

Testing *may* be done on the development deployment, but care should be taken if the tests should be performed on every commit. There are several factors, which may be disruptive:

1. A developer may forget or ignore installing client-side git hooks,
2. The tests may be long-running and would prohibit a high-volume git commit workflow

It's recommended to move the testing to the server side and test after some code is pushed. Again, if a failing test is detected - it shouldn't be a cause for stopping work, it should just be a cause

[13] https://jenkins.io
[14] https://travis-ci.org/
[15] https://buildkite.com
[16] https://codeship.com

of not performing a deployment. Developers should have the capability of running tests on their development environment, give them this option by providing an individual “test.sh” or some documented way to do this.

With a Go project, this might be as simple as running `go test` but your application may test other things as well - like functionality with modifying database values. These tests can be more elaborate and use Docker for setting up a database container, loading testing data and then running parts of your application as if they were in a production system.

> Keep a practice of not assigning blame - when something breaks, just assign who is responsible for a fix. You should keep accountability while removing blame. Keep your team involved, so you may foster a team culture in resolving these issues. With individual issues, pair the person responsible with a reviewer, so that they may together decide on a joint solution of an issue. Accept the issue, learn from it, and solve it - creating a problem is a learning opportunity.

II. Dependencies - Explicitly declare and isolate dependencies

Go includes a packaging system for installing packages, `go get`. While you can install packages your application uses this way, you have no control over which package will be installed when you run it. As you want to have a repeatable build process, using `go get` is not advised, but you should look to vendoring to find an alternative.

Vendoring

For you guys coming from PHP, Python or Node: there you have `composer`, `pip` and `npm`. For Go, you have a wider choice of package management tools[17] which create a vendor/ folder. I personally recommend using gvt[18], other people also suggested glide[19].

Fetching a dependency with gvt

```
gvt fetch github.com/namsral/flag
```

Using `gvt fetch` is the equivalent of using `go get`, with the difference that the resulting package is written in the `vendor/` folder. You should commit this folder to your VCS, keeping it there for future builds. If you need to update the dependencies for some reason, you can at a later time issue `gvt update`, and then commit the changes to the vendor/ folder.

You can list your current dependencies by issuing `gvt list`.

```
$ gvt list
github.com/namsral/flag https://github.com/namsral/flag master 881a43080604bcf99ab11\
18a814d1cb2c268fc36
```

If you don't want to keep the packages committed to your source repository, you can add this `.gitignore` rule:

[17] https://github.com/golang/go/wiki/PackageManagementTools
[18] https://github.com/FiloSottile/gvt
[19] https://github.com/Masterminds/glide

```
vendor/**
!vendor/manifest
```

This will allow you only to commit the manifest file to your git repository, without any other sources. When you want to make a clean build of your application, you would check out your sources and issue `gvt restore` to fetch all the dependencies declared in the manifest file. You can also use `gvt update` to stay at the tip of your dependencies.

> Note: when using this approach, `gvt` is required in the build environment. Your build may fail because of network connectivity issues, renaming or deleting any of the upstream packages. If you commit your complete vendor folder, it guarantees that you will be able to reproduce the build at any later time.

Caveats about vendoring

Vendoring is an advocated way to provide packages to your application, with more fine grained control as to what version or branch of the package you would like to use. That being said, there are a few points in the small print which you need to consider.

Dependencies

Your code, subpackages or vendored packages might depend on different versions of the same package. This usually means that you'll need to fix some code, so that they can work on the same version, or that you will need to provide some kind of additional versioning for your own packages.

A project which aims to resolve a part of this problem is gopkg.in[20]. When you're importing dependencies, you can specify a tag or branch by constructing an URL which includes this information. There are two url patterns supported.

```
gopkg.in/pkg.v3      â†’ github.com/go-pkg/pkg (branch/tag v3, v3.N, or v3.N.M)
gopkg.in/user/pkg.v3 â†’ github.com/user/pkg   (branch/tag v3, v3.N, or v3.N.M)
```

Unfortunately, your dependency needs to provide branches/tags to support this. Some of them do. If yours doesn't, you might have to fork the original repository, and add some tags or branches, so you can separate the versions.

You can also leverage `gvt` to do this, but you might have to edit the manifest file yourself, as this is not a common use-case. When you do it, keep in mind that `gvt update` will update all packages to the latest revision - this might be problematic, if you're bound to a specific tag or commit.

Relative imports

In your application, you may use something like this:

[20] http://labix.org/gopkg.in

```
import "./subpackage"
```

While this is a perfectly valid pattern, it doesn't work with vendoring. If your package is using any kind of vendored imports, your application will not build. If your application lives in `$GOPATH/src/app` as it should, use this import instead:

```
import "app/subpackage"
```

This import searches for files in the `$GOPATH/src/app/subpackage` folder, which is exactly where your files are. The reason why you should do this is that relative imports are currently not working correctly with vendoring. People are suggesting to avoid relative imports altogether, but I suspect this may be fixed in a future version. It makes perfect sense for relative paths under your project folder, but not above (with `../`).

> You could use tooling to rewrite parts of your source to change patterns like "./" to "app/", but you'd have to use it instead of `go run` and `go build` as a pre-processing step. While this could be acceptable in your build environment, it would be a hassle to do it in your development environment. My suggestion is to stick to the default tools here - there's no benefit in additional tooling which you have to develop and maintain, unless it provides strong value.

System dependencies

When building docker images, you use the Dockerfile to declare your dependencies. A very minimal Dockerfile for creating a `bash` docker image would look like this:

```
FROM alpine
RUN apk add --no-cache bash
```

This serves as a *declaration* of a dependency for your application. By building a docker image, the image itself is *isolated* from other containers. The equivalent image on a Debian system might look something like this.

```
FROM debian
RUN apt-get update && apt-get install bash
```

Ah, it sounds funny, but the RUN command isn't actually needed, as Debian already comes packaged with `bash`. It would theoretically update bash to the latest version, if you're not updating the FROM source image.

When it comes to building docker images, you can compose your images this way. For example, there's an official package for `golang:1.8-alpine`, which you would use for `go run` and `go build` commands. If you wanted to create a "gvt" image, you would create a new Dockerfile, and add on something like:

```
FROM golang:1.8-alpine
RUN go get -u github.com/FiloSottile/gvt
```

You would repeat this, for every tool which is installed with `go get`, creating a new image for each of them. Or you could just declare one "big" Go image like this:

```
FROM golang:1.8-alpine

MAINTAINER Tit Petric <black@scene-si.org>

## install needed packages
RUN  apk --no-cache add gcc musl-dev git

## add external get packages
RUN  go get -u github.com/FiloSottile/gvt && \
  go get -u github.com/Masterminds/glide && \
  go get -u github.com/kisielk/errcheck && \
  go get -u github.com/golang/lint/golint  && \
  go get -u github.com/goreleaser/goreleaser
```

This is already considered bad practice, because you don't have explicit control over individual applications. If you want to have `gvt`, you should build an image which provides only gvt. This image would be based in a parent golang image.

I don't agree on this point very strongly - the single responsibility principles with building docker images to best practices are dealing with running parts. With such an image, the only running part is the CLI program which you run - go, gvt, glide, or any other from the list.

The problem of violating single responsibility principles comes where you have for example, supervisord, nginx, and php-fpm running in the same container. You can't change php-fpm without modifying all three. The idea is that you only modify one, and this means less possibility of human error (or just isolated to one service, instead of all of them).

> I'm going to play devils advocate here and say that exposing php on a FastCGI interface (TCP) is a bit of a long-stretch. It would depend on the knowledge of the person operating it, but HTTP in comparison is a better understood protocol where people have been known to craft their simple queries (i.e. `GET / HTTP/1.0`) to inspect that something works. FastCGI interfaces are hidden away (on either unix sockets or TCP) and don't provide introspection tools - like the Developer console you're using in your favourite browser. There are of course valid reasons why these services *should* be separated, but alas, they end up together many times for convenience reasons.

Depending on your build/release/run stages, you will most likely separate your dependencies on those. For example, in your build stage you may need two different versions of Go, you may use the official golang[21] image with the respective version tag (1.8 currently). But for your release/run stages, the `golang` dependency is not required, as you're already dealing with a compiled application binary.

You may have similar development-only requirements for other software, like `npm`, `pip`, `composer`, `yarn`, `webpack` and many others. Include what you like, but choose carefully.

In some cases the simplest way to add a dependency is to issue a `git clone`, while in other cases you might be better off using what is available on Docker Hub (or Alpine/Debian/… package mirrors). If you add all your dependencies to your application development Dockerfile, it will inevitably take a very long time to build.

When we will talk about backing services, you should know that there are a number of "official" docker images available on the Docker Hub[22] and you may choose between them to use, for example `percona:5.5`, `percona:5.7`, or extend or build any image that fits your requirements.

It's not uncommon for people to build their packages from sources, but it does take additional effort to keep the size of such builds in check. Versioned docker images also serve not only as a declaration of dependency, but also as a possible way to roll back your releases (more about that in Chapter V.).

[21] https://hub.docker.com/_/golang/
[22] https://hub.docker.com/explore/

When creating your Dockerfiles, please follow the single responsibility principle[23]. If your docker image is self-contained, migration for it between hosts is easily accomplished. When your docker image contains multiple services, you might have to deal with several graceful shutdowns to effectively migrate data, which will introduce operational complexity and risks. If you do this, you are also clearly violating several chapters of 12FA, most notably dependency isolation from this chapter, and chapter IX. Disposability.

Depending on your development patterns, you should pay attention to provide small Docker images. I tend to travel and work from countries with poor or limited internet connectivity, where every megabyte saved can improve my productivity drastically. I vividly remember downloading images about 100MB in size for about 15 minutes. If you can make a coffee and drink it before the image finishes downloading, that's a good hint that you should optimize it.

[23] https://en.wikipedia.org/wiki/Single_responsibility_principle