The Twelve-Factor App is a recent methodology (and/or a manifesto) for writing web applications that, hopefully, is getting quite popular. Although I don't agree 100% with the recommendations, I'll quickly go through all 12 factors and discuss them from the perspective of the Java ecosystem, mentioning the absolute "musts" and the points where I disagree. For more, visit the 12factor.net site.

1. **Codebase – one codebase, multiple deploys**. **This means you must not have various codebase for various versions. Branches is okay, different repositories are not**. I'd even go further and not recommend Subversion. Not because it's not fine, but because git and mercurial do the same and much more. You can use git/mercurial the way you use SVN, but not the other way around. And tools for DVCS (e.g. SourceTree) are already quite good.

2. **Dependencies – obviously, you must put as many dependencies in your manifests (e.g. pom.xml) as possible.** The manifesto advices against relying on pre-installed software, for example ImageMagick or Pandoc, but I wouldn't be that strict. If your deployments are automated and you guarantee the presence of a given tool, you shouldn't spend days trying to wrap it in a library of your working language. If it's as easy as putting an exetuable script in a jar file and then extracting it, that's fine. But if it requires installation, and you really need it (ImageMagick is a good example indeed), I don't think it's wrong to expect it to be installed. Just check on startup if it's present and fail fast if it's not.

3. **Config – the most important rule here is – never commit your environment-specific configuration (most importantly: password) in the source code repo**. Otherwise your production system may be vulnerable, as are probably at least a third of these wordpress deployments (and yes, mysql probably won't allow external connections, but I bet nobody has verified that). But from there on my opinion is different than the one of the 12-factor app. No, you shouldn't use environment variables for your configuration. Because when you have 15 variables, managing them becomes way easier if they are in a single file. You can have some shell script that sets them all, but that goes against the OS independence. Having a key-value .properties file (for which Java has native support), and only passing the absolute path to that file as an environment variable (or JVM param) is a better approach, I think. I've discussed it previously. E.g. CONFIG_PATH=/var/conf/app.properties, which you load on startup. And in your application you can keep a blank app.example.properties which contains a list of all properties to be configured – database credentials, keys and secrets for external systems, etc. (without any values). That way you have all the properties in one place and it's very easy to discover what you may need to add/reconfigure in a given scenario. If you use environment variables, you'd have to have a list of them in a txt file in order to make them "discoverable", or alternatively, let the developers dig into the code to find out which properties are available. And last, but not least – when I said that you shouldn't commit properties files to source control, there is one very specific exception. You can choose to version your environment configurations. It must be a private repo, with limited access and all that, but the (Dev)Ops can have a place where they keep the properties and other specifics for each environment, versioned. It's easier to have that with a properties file (not impossible with env variables, but then again you need a shell script). The 12-factor app authors warn about explosion of environments. If you have a properties file for each environment, these may grow. But they don't have to. You can change the values in a properties file exactly the way you would manage the environment variables.

4. **Backing Services – it's about treating that external services that your application depends on equally, regardless of whether you manage them, or whether another party manages them.** From the application's perspective that should not matter. What I can add here is that you should try

to minimize this. If an in-memory queue would do, don't deploy a separate MQ. If an in-memory cache would do, don't deploy a redis instance. If an embedded database would do, don't manage a DB installation (e.g. neo4j offers an embedded variant). And so on. But if you do need the full-featured external service, make the path/credentials to it configurable as if it's external (rather than, for example, pointing to localhost by default).

5. **Build, release, run** – it is well described on the page. It is great to have such a lifecycle. But it takes time and resources to set it up. Depending on your constraints, you may not have the full pipeline, and some stages may be more manual and fluid than ideal. Sometimes, for example in the early stages of a startup, it may be beneficial to be able to swap class files or web pages on a running production server, rather than going through a full release process (which you haven't had the time to fully automate). I know this sounds like heresy, and one should strive to a fully automated and separated process, but before getting there, don't entirely throw away the option for manually dropping a fixed file in production. As longs as you don't do it all the time and you don't end up with a production environment for which you have no idea what version of the codebase is run.

6. **Processes – this is about being stateless, and also about not relying on any state being present in memory or on the file system. And indeed, state does not belong in the code**. However, there's something I don't agree with. The 12-factor preferred way of packaging your assets is during build time (merging all css files into one, for example). That has several drawbacks – you can't combine assets dynamically, e.g. if you have 6 scripts, and on one page you need 4, on another page you need 2 of the ones used on the first page, and another 2, then you have to build all this permutations beforehand. Which is fine and works, but why is it needed? There is no apparent benefit. And depending on the tools you use, it may be easier to work with CDN if you are dynamically generating the bundles. Another thing where further Java-related details can be given is "sticky sessions". It's not a good idea to have them, but note that you can use your session to store data about the user in memory. You just have to configure your servlet container (or application server) to share that state. Basically, under the hood it still uses a distributed cache like memcached or ehcache (I guess you could also use a redis implementation of the session clustering). It's just transparent from the developer and he can still use the session store.

7. **Port Binding – this is about having your application as standlone, instead of relying on a running instance of an application server, where you deploy.** While that seems easier to manage, it isn't. Starting an servlet container and pushing a deployment is just as easy. But in order to have your application bind to a port, you need to have the tooling for that. They mention jetty, and there is also an embedded version of tomcat, and spring-boot (which wraps both). And while I'm not against the port binding, I'd say it's equally good to have it the other way around. Container configuration is done equally easy, regardless of whether you drop an environment-specific xml file, or do it programmatically and load the properties from the file mentioned in point 3. The point is – it doesn't matter – do whichever is easier for you. Not to mention that you may need some apache/nginx functionality.

8. **Concurrency** – it's about using native processes. This, I think, isn't so relevant to a Java runtime, which uses threads under the hood and hides away the unix process. By the way, another explicit reference to unix (rather than staying OS-independent).

9. **Disposability – that's about embracing failure. Your system must work fine even though one or more of application instances die.** And that's bound to happen, especially "in the cloud". They

mention SIGTERM, which is a *nix-specific signal, whereas the general idea of the 12-factor app is to be OS-independent. There is an apparent leaning towards Linux, which is fine though.

10. **Dev/prod parity – your development environment should almost identical to a production one (for example, to avoid some "works on my machine" issues)**. That doesn't mean your OS has to be the OS running in production, though. You can run Windows, for example, and have your database, MQ, etc. running on a local virtual machine (like my setup). This also underlines the OS-independence of your application. Just have in mind to keep the versions the same.

11. **Logs – the 12-factor app recommends writing all logging information to the system out. A Java developer will rightly disagree. With tools like loggack/slf4j you can manage the logging aspects within the application, rather than relying on 3rd party tools to do that. E.g. log rotation and cleanup, or sending to a centralized logging facility.** It's much easier to configure a graylog or splunk adapter, than having another process gather that from system out and push it. There can be environment-specific log configurations, which is again just one file bundled together with the app.properties). If that seems complicated, consider the complications of setting up whatever is going to capture the output.

12. **Admin processes** – generally agreed, but in addition I'd say it's preferable to execute migrations on deployment or startup, rather than manually, and that manually changing "stuff" on production should preferably be done through something like capistrano in order to make sure it's identical on all instances.

Overall, it's a good set of advice and an approach to building apps that I'd recommend, with the above comments in mind.