The Twelve Factors - 2022

**Remember**

🡺 **Codebase, Dependencies, Config Logs Backing services**

🡺 **Admin Processes Concurrency Disposability**

🡺 **Build, release, run via Port binding with Dev/prod parity**

**Codebase:** One codebase tracked in revision control, many deploys

**Dependencies**: Explicitly declare and isolate dependencies

**Config**: Store config in the environment

**Logs**: Treat logs as event streams

**Backing services**: Treat backing services as attached resources

**Admin processes**: Run admin/management tasks as one-off processes

**Processes**: Execute the app as one or more stateless processes

**Concurrency**: Scale out via the process model

**Disposability**: Maximize robustness with fast startup and graceful shutdown

**Build, release, run**: Strictly separate build and run stages

**Port binding**: Export services via port binding

**Dev/prod parity**: Keep development, staging, and production as similar as possible

**Codebase: One codebase tracked in revision control, many deploys**.

As per 12-factor app, every application should have its own codebase (repos). Multiple apps sharing the same code are a violation of the twelve-factor. In Microservices, every service should have its own codebase.

**Dependencies**: **Explicitly declare and isolate dependencies**

The **twelve-factor app should always explicitly declare all its dependencies**. We should do this using a dependency declaration manifest. Java has multiple dependency management tools like Maven and Gradle. **All the application packages will be managed through package managers like sbt, maven, Gradle.**

**Config**: **Store config in the environment**

**Anything that varies between the deployment environments is considered as configuration**. This includes **Database connections** and **credentials**, **system integration endpoints**, Application-specific information like IP Addresses, ports, and hostnames. **Externalize the configurations from the application**. you can manage the configurations for your applications from a source control like git (spring-cloud-config) and use the environment variables.

**Logs**: **Treat logs as event streams**

Twelve-factor app principles advocate **separating the log generation and processing the log's information**. In Microservices, observability is the first-class citizen. Observability can be achieved through using APM tools (**ELK**, Newrelic, and other tools) or log aggregations tools **like Splunk, logs**, etc.

**Backing services**: **Treat backing services as attached resources**

A backing service is an application/service the app consumes over the network as part of its normal operation. Database, Message Brokers, any other external systems can be treated as Backing service.

**12-factor app can automatically swap the application from one provider to another without making any further modifications to the code base**. Let us say, you would like to change the database server from MySQL to Aurora. To do so, you should not make any code changes to your application. Only configuration change should be able to take care of it. In a microservice ecosystem, anything external to service is treated as attached resource.

**Admin Process**: **Run admin/management tasks as one-off processes**

**What is one-off process**? This principle describes that **administrative or management tasks should be executed as separate short-lived processes or ad-hoc process**. **One-off tasks** can be independent of any process or can be added to any active processes. **One-off admin processes should be run in an identical environment**. They run against a [release](https://12factor.net/build-release-run), using the same [codebase](https://12factor.net/codebase) and [config](https://12factor.net/config) as any process run against that release. Admin code must ship with application code to avoid synchronization issues. **Twelve-factor principles advocates for keeping such administrative tasks as part of the application codebase in the repository**. There is a number of one-off processes as part of the application deployment like data migration, executing one-off scripts in a specific environment.

**Processes**: **Execute the app as one or more stateless processes**

**A twelve-factor app is expected to run in an execution environment as stateless processes.**In other words, they cannot store persistent state locally between requests. Some web systems rely on [“sticky sessions”](http://en.wikipedia.org/wiki/Load_balancing_%28computing%29#Persistence) – that is, caching user session data in memory of the app’s process and expecting future requests from the same visitor to be routed to the same process. **Sticky sessions are a violation of twelve-factor and should never be used or relied upon**. Session state data is a good candidate for a datastore that offers time-expiration, such as [Memcached](http://memcached.org/) or [Redis](http://redis.io/)**.**

**Concurrency**: **Scale out via the process model**

Twelve-factor app processes [should never daemonize](http://dustin.github.com/2010/02/28/running-processes.html) or write PID files. This talks about scaling the application. Twelve-factor app principles suggest to consider running your application as multiple processes/instances instead of running in one large system. You can still opt-in for threads to improve the concurrent handling of the requests. In a nutshell, twelve-factor app principles advocate for **horizontal scaling** instead of vertical scaling.

Vertical scaling 🡺 Add additional hardware to the system

Horizontal scaling 🡺 Add additional instances of the application

**Disposability**: **Maximize robustness with fast startup and graceful shutdown**

The twelve-factor app's processes are disposable, meaning they can be started or stopped at a moment's notice. When the application is shutting down or starting, an instance should not impact the application state. Graceful shutdowns are very important. The system must ensure the correct state.

The system should not get impacted when new instances are added or takedown the existing instances as per need. This is also known as system disposability.

**Build, release, run**: **Strictly separate build and run stages**

The application must have a strict separation between the build, release, and run stages. Let us understand each stage in more detail.

* **Build stage:**transform the code into an executable bundle/ build package.
* **Release stage:** get the build package from the build stage and combines with the configurations of the deployment environment and make your application ready to run.
* **Run stage:** It is like running your app in the execution environment.

You can use CI/CD tools to automate the builds and deployment process. Docker images make it easy to separate the build, release, and run stages more efficiently.

**Port binding: Export services via port binding**

**The twelve-factor app is completely self-contained and doesn't rely on runtime injection of a webserver** into the execution environment to create a web-facing service. The web app exports HTTP as a service by binding to a port, and listening to requests coming in on that port.

**In short, this is all about having your application as a standalone instead of deploying them into any of the external web servers.** Spring boot is one example of this one. Spring boot by default comes with embedded tomcat, jetty, or undertow.

**Dev/prod parity**: **Keep development, staging, and production as similar as possible**

The twelve-factor methodology suggests keeping the **gap between development and production** environment as minimal as possible. This reduces the risks of showing up bugs in a specific environment. Now, technology like Spring Boot and Docker automatically bridge this gap to a great extent. A containerized application is expected to behave the same, no matter where we run it.

A codebase is any single repo (in a centralized revision control system like Subversion). There is always a one-to-one correlation between the codebase and the app.

* If there are multiple codebases, it’s not an app – it’s a distributed system. Each component in a distributed system is an app, and each can individually comply with twelve-factor.
* Multiple apps sharing the same code is a violation of twelve-factor. The solution here is to factor shared code into libraries which can be included through the [dependency manager](https://12factor.net/dependencies).

There is only one codebase per app, but there will be many deploys of the app. A *deploy* is a running instance of the app. This is typically a production site, and one or more staging sites. Additionally, every developer has a copy of the app running in their local development environment, each of which also qualifies as a deploy. The codebase is the same across all deploys, although different versions may be active in each deploy.

## II. Dependencies: Explicitly declare and isolate dependencies

The **twelve-factor app should always explicitly declare all its dependencies**. We should do this using a dependency declaration manifest. Java has multiple dependency management tools like Maven and Gradle. We can use one of them to achieve this goal. Twelve-factor apps also do not rely on the implicit existence of any system tools. Examples include shelling out to ImageMagick or curl. While these tools may exist on many or even most systems, there is no guarantee that they will exist on all systems where the app may run in the future, or whether the version found on a future system will be compatible with the app. If the app needs to shell out to a system tool, that tool should be vendored into the app.

## III. Config: Store config in the environment

An app’s config is everything that is likely to vary between [deploys](https://12factor.net/codebase) (staging, production, developer environments, etc). This includes:

* Resource handles to the database, Memcached, and other [backing services](https://12factor.net/backing-services)
* Credentials to external services such as Amazon S3 or Twitter
* Per-deploy values such as the canonical hostname for the deploy

Apps sometimes store config as constants in the code. This is a violation of twelve-factor, which requires **strict separation of config from code**. Config varies substantially across deploys, code does not. **The twelve-factor app stores config in *environment variables***

**A twelve-factor app should externalize all such configurations that vary between deployments**. The recommendation here is to use environment variables for such configurations. This leads to a clean separation of config and code.

Spring provides a configuration file where we can declare such configurations and attach it to environment variables:

spring.datasource.url=jdbc:mysql://${MYSQL\_HOST}:${MYSQL\_PORT}/movies

spring.datasource.username=${MYSQL\_USER}

spring.datasource.password=${MYSQL\_PASSWORD}

On Windows, we can set the environment variable before starting the application:

set MYSQL\_HOST=localhost

set MYSQL\_PORT=3306

set MYSQL\_USER=movies

set MYSQL\_PASSWORD=password

## IV. Backing services: Treat backing services as attached resources

### A backing service is any service the app consumes over the network as part of its normal operation. Examples include datastores (such as MySQL or CouchDB), messaging/queueing systems (such as RabbitMQ or Beanstalkd). A *backing service* is any service on which your application relies for its functionality. It can be database, messaging etc. The Backing Services principle encourages architects to treat external components such as databases, email servers, message brokers, and independent services that can be provisioned and maintained by systems personnel as attached resources.

**The code for a twelve-factor app makes no distinction between local and third party services.**

**A twelve-factor app should treat all such backing services as attached resources.**

In our application, we've used [MySQL](https://www.mysql.com/) as the backing service to provide persistence.

[Spring JPA](https://www.baeldung.com/the-persistence-layer-with-spring-and-jpa) makes the code quite agnostic to the actual database provider. We only need to define a repository which provides all standard operations:

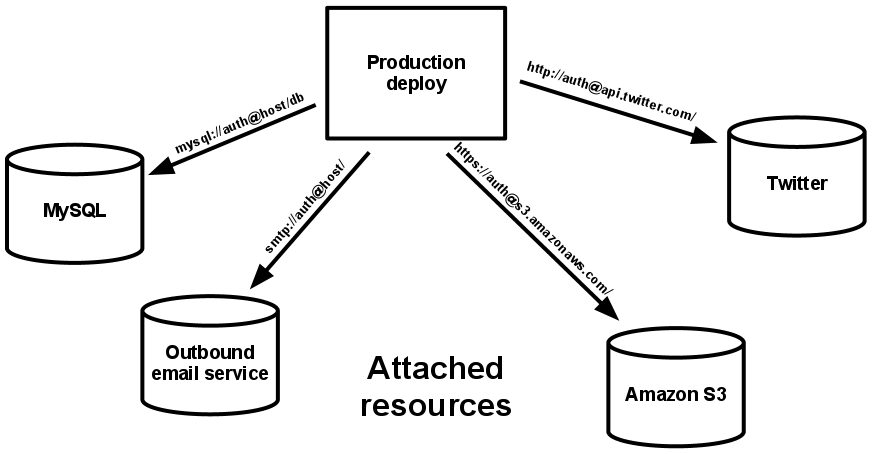
@Repository

**public** **interface** **MovieRepository** **extends** **JpaRepository**<**Movie**, **Long**> {

}

As we can see, this is not dependent on MySQL directly. Spring detects the MySQL driver on the classpath and provides a MySQL-specific implementation of this interface dynamically. Moreover, it pulls other details from configurations directly.

So, if we've to change from MySQL to Oracle, all we've to do is replace the driver in our dependencies and replace the configurations.



## V. Build, release, run: Strictly separate build and run stages

A [codebase](https://12factor.net/codebase) is transformed into a (non-development) deploy through three stages:

* The *build stage* is a transform which converts a code repo into an executable bundle known as a *build*. Using a version of the code at a commit specified by the deployment process, the build stage fetches vendors [dependencies](https://12factor.net/dependencies) and compiles binaries and assets.
* The *release stage* takes the build produced by the build stage and combines it with the deploy’s current [config](https://12factor.net/config). The resulting *release* contains both the build and the config and is ready for immediate execution in the execution environment.
* The *run stage* (also known as “runtime”) runs the app in the execution environment, by launching some set of the app’s [processes](https://12factor.net/processes) against a selected release.

**The twelve-factor app uses strict separation between the build, release, and run stages.**

Every release should always have a unique release ID, such as a timestamp of the release (such as 2011-04-06-20:32:17) or an incrementing number (such as v100). Releases are an append-only ledger and a release cannot be mutated once it is created. Any change must create a new release.

The twelve-factor methodology **strictly separates the process of converting codebase into a running application** as three distinct stages:

* Build Stage: This is where we take the codebase, perform static and dynamic checks, and then generate an executable bundle like a JAR. Using a tool like [**Maven**](https://maven.apache.org/), this is quite trivial:

mvn clean compile test package

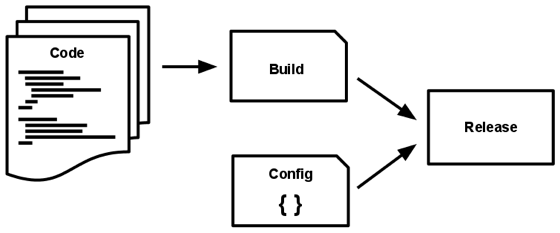
* Release Stage: This is the stage where we take the executable bundle and combine this with the right configurations. Here, we can use [**Packer**](https://www.packer.io/) with a provisioner like [**Ansible**](https://www.ansible.com/) to create Docker images:

packer build application.json

* Run Stage: Finally, this is the stage where we run the application in a target execution environment. If we use [**Docker**](https://www.docker.com/) as the container to release our application, running the application can be simple enough:

docker run --name <container\_id> -it <image\_id>

Finally, we don't necessarily have to perform these stages manually. This is where [Jenkins](https://jenkins.io/) comes in as pretty handy with their declarative pipeline.



## VI. Processes: Execute the app as one or more stateless processes

**The app is executed in the execution environment as one or more *processes*.**

In the simplest case, the code is a stand-alone script, the execution environment is a developer’s local laptop with an installed language runtime, and the process is launched via the command line (for example, python my\_script.py). On the other end of the spectrum, a production deploy of a sophisticated app may use many [process types, instantiated into zero or more running processes](https://12factor.net/concurrency).

**Twelve-factor processes are stateless and**[**share-nothing**](http://en.wikipedia.org/wiki/Shared_nothing_architecture)**.** Any data that needs to persist must be stored in a stateful [backing service](https://12factor.net/backing-services), typically a database.

Some web systems rely on [“sticky sessions”](http://en.wikipedia.org/wiki/Load_balancing_%28computing%29#Persistence) – that is, caching user session data in memory of the app’s process and expecting future requests from the same visitor to be routed to the same process. **Sticky sessions are a violation of twelve-factor and should never be used or relied upon**. Session state data is a good candidate for a datastore that offers time-expiration, such as [Memcached](http://memcached.org/) or [Redis](http://redis.io/)**.**

**A twelve-factor app is expected to run in an execution environment as stateless processes.**In other words, they can not store persistent state locally between requests. They may generate persistent data which is required to be stored in one or more stateful backing services.

In the case of our example, we've got multiple endpoints exposed. A request on any of these endpoints is entirely independent of any request made before it. For instance, if we keep track of user requests in-memory and use that information to serve future requests, it violates a twelve-factor app.

## VII. Port binding: Export services via port binding

The twelve-factor app is completely self-contained and does not rely on runtime injection of a webserver into the execution environment to create a web-facing service. The web app exports HTTP as a service by binding to a port, and listening to requests coming in on that port.

In a local development environment, the developer visits a service URL like **http://localhost:5000/** to access the service exported by their app. In deployment, a routing layer handles routing requests from a public-facing hostname to the port-bound web processes.

A traditional web application in Java is developed as a WAR or web archive. This is typically a collection of Servlets with dependencies, and it expects a conformant container runtime like Tomcat.**A twelve-factor app, on the contrary, expects no such runtime dependency.** It's completely self-contained and only requires an execution runtime like Java.

In our case, we've developed an application using Spring Boot. Spring Boot, apart from many other benefits, provides us with a default embedded application server. Hence, the JAR we generated earlier using Maven is fully capable of executing in any environment just by having a compatible Java runtime:

java -jar application.jar

Here, our simple application exposes its endpoints over an HTTP binding to a specific port like 8080. Upon starting the application as we did above, it should be possible to access the exported services like HTTP. An application may export multiple services like FTP or [WebSocket](https://www.baeldung.com/websockets-spring) by binding to multiple ports.

## VIII. Concurrency: Scale out via the process model

### In the twelve-factor app, processes are a first class citizen. Processes in the twelve-factor app take strong cues from [the unix process model for running service daemons](https://adam.herokuapp.com/past/2011/5/9/applying_the_unix_process_model_to_web_apps/). Using this model, the developer can architect their app to handle diverse workloads by assigning each type of work to a *process type*. For example, HTTP requests may be handled by a web process, and long-running background tasks handled by a worker process.

The process model truly shines when it comes time to scale out. The [share-nothing, horizontally partitionable nature of twelve-factor app processes](https://12factor.net/processes) means that adding more concurrency is a simple and reliable operation. The array of process types and number of processes of each type is known as the *process formation*.

Twelve-factor app processes [should never daemonize](http://dustin.github.com/2010/02/28/running-processes.html) or write PID files. Instead, rely on the operating system’s process manager (such as [systemd](https://www.freedesktop.org/wiki/Software/systemd/), a distributed process manager on a cloud platform, or a tool like [Foreman](http://blog.daviddollar.org/2011/05/06/introducing-foreman.html) in development) to manage [output streams](https://12factor.net/logs), respond to crashed processes, and handle user-initiated restarts and shutdowns.

**The twelve-factor methodology suggests apps to rely on processes for scaling.**

What this effectively means is that applications should be designed to distribute workload across multiple processes. Individual processes are, however, free to leverage a concurrency model like Thread internally.

A Java application, when launched gets a single process which is bound to the underlying JVM. What we effectively need is a way to launch multiple instances of the application with intelligent load distribution between them. Since we've already packaged our application as a [Docker](https://www.baeldung.com/docker-java-api) container, [Kubernetes](https://www.baeldung.com/kubernetes) is a natural choice for such orchestration.

## IX. Disposability: Maximize robustness with fast startup and graceful shutdown

**The twelve-factor app’s**[**processes**](https://12factor.net/processes)**are *disposable*, meaning they can be started or stopped at a moment’s notice.** This facilitates fast elastic scaling, rapid deployment of [code](https://12factor.net/codebase) or [config](https://12factor.net/config) changes, and robustness of production deploys.

Processes should strive to **minimize startup time**. Short startup time provides more agility for the [release](https://12factor.net/build-release-run) process and scaling up; and it aids robustness, because the process manager can more easily move processes to new physical machines when warranted.

Processes **shut down gracefully when they receive a**[**SIGTERM**](http://en.wikipedia.org/wiki/SIGTERM) signal from the process manager. For a web process, graceful shutdown is achieved by ceasing to listen on the service port (thereby refusing any new requests), allowing any current requests to finish, and then exiting. Implicit in this model is that HTTP requests are short (no more than a few seconds), or in the case of long polling, the client should seamlessly attempt to reconnect when the connection is lost. Processes should also be **robust against sudden death**, in the case of a failure in the underlying hardware.

Application processes can be shut down on purpose or through an unexpected event. In either case, **a twelve-factor app is supposed to handle it gracefully**. In other words, an application process should be completely disposable without any unwanted side-effects. Moreover, processes should start quickly

For instance, in our application, one of the endpoints is to create a new database record for a movie. Now, an application handling such a request may crash unexpectedly. This should, however, not impact the state of the application. When a client sends the same request again, it shouldn't result in duplicate records.

In summary, the application should expose [idempotent](https://www.baeldung.com/cs/idempotent-operations) services. This is another very desirable attribute of a service destined for cloud deployments. This gives the flexibility to stop, move, or spin new services at any time without any other considerations.

## X. Dev/prod parity:Keep development, staging, and production as similar as possible

### The twelve-factor app is designed for [continuous deployment](http://avc.com/2011/02/continuous-deployment/) by keeping the gap between development and production small. Looking at the three gaps described above:

* Make the time gap small: a developer may write code and have it deployed hours or even just minutes later.
* Make the personnel gap small: developers who wrote code are closely involved in deploying it and watching its behavior in production.
* Make the tools gap small: keep development and production as similar as possible.

**The twelve-factor methodology suggests keeping the gap between development and production environment as minimal as possible.** These gaps can result from long development cycles, different teams involved, or different technology stack in use.

Now, technology like Spring Boot and Docker automatically bridge this gap to a great extent. A containerized application is expected to behave the same, no matter where we run it. We must use the same backing services – like the database – as well.

Moreover, we should have the right processes like continuous integration and delivery to facilitate bridging this gap further.

## XI. Logs: Treat logs as event streams

### Logs are the [stream](https://adam.herokuapp.com/past/2011/4/1/logs_are_streams_not_files/) of aggregated, time-ordered events collected from the output streams of all running processes and backing services.

**A twelve-factor app never concerns itself with routing or storage of its output stream.** It should not attempt to write to or manage logfiles. Instead, each running process writes its event stream, unbuffered, to stdout. During local development, the developer will view this stream in the foreground of their terminal to observe the app’s behavior.

The event stream for an app can be routed to a file, or watched via realtime tail in a terminal. Most significantly, the stream can be sent to a log indexing and analysis system such as [Splunk](http://www.splunk.com/), or a general-purpose data warehousing system such as [Hadoop/Hive](http://hive.apache.org/).

These systems allow for great power and flexibility for introspecting an app’s behavior over time, including:

* Finding specific events in the past.
* Large-scale graphing of trends (such as requests per minute).
* Active alerting according to user-defined heuristics (such as an alert when the quantity of errors per minute exceeds a certain threshold).

A twelve-factor app, however, separates itself from log generation and its processing. **For such an app, logs are nothing but a time-ordered stream of events.** It merely writes these events to the standard output of the execution environment. The capture, storage, curation, and archival of such stream should be handled by the execution environment.

There are quite several tools available to us for this purpose. To begin with, we can use [SLF4J](https://www.baeldung.com/slf4j-with-log4j2-logback) to handle logging abstractly within our application. Moreover, we can use a tool like [Fluentd](https://www.fluentd.org/) to collect the stream of logs from applications and backing services.

This we can feed into [Elasticsearch](https://www.elastic.co/) for storage and indexing. Finally, we can generate meaningful dashboards for visualization in [Kibana](https://www.elastic.co/products/kibana).

## XII. Admin processes: Run admin/management tasks as one-off processes

The [process formation](https://12factor.net/concurrency) is the array of processes that are used to do the app’s regular business (such as handling web requests) as it runs. Separately, developers will often wish to do one-off administrative or maintenance tasks for the app, such as:

* Running database migrations (e.g. manage.py migrate in Django, rake db:migrate in Rails).
* Running a console (also known as a [REPL](http://en.wikipedia.org/wiki/Read-eval-print_loop) shell) to run arbitrary code or inspect the app’s models against the live database. Most languages provide a REPL by running the interpreter without any arguments (e.g. python or perl) or in some cases have a separate command (e.g. irb for Ruby, rails console for Rails).
* Running one-time scripts committed into the app’s repo (e.g. php scripts/fix\_bad\_records.php).

### One-off admin processes should be run in an identical environment as the regular [long-running processes](https://12factor.net/processes) of the app. They run against a [release](https://12factor.net/build-release-run), using the same [codebase](https://12factor.net/codebase) and [config](https://12factor.net/config) as any process run against that release. Admin code must ship with application code to avoid synchronization issues.

Often we need to perform some one-off tasks or routine procedure with our application state. For instance, fixing bad records. Now, there are various ways in which we can achieve this. Since we may not often require it, we can write a small script to run it separately from another environment.

Now, **the twelve-factor methodology strongly suggests keeping such admin scripts together with the application codebase**. In doing so, it should follow the same principles as we apply to the main application codebase. It's also advisable to use a built-in REPL tool of the execution environment to run such scripts on production servers.

In our example, how do we seed our application with the already watched movies so far? While we can use our sweet little endpoint, but that may seem to be impractical. What we need is a script to perform a one-time load. We can write a small Java function to read a list of movies from a file and save them in batch into the database. Moreover, we can use [Groovy integrated with Java](https://www.baeldung.com/groovy-java-applications) runtime to start such processes.

# 12 Factors and Beyond in Java

## 1. One Codebase

While less of a Java-specific concept, this factor generally refers to getting to a single code base managed in source control or a set of repositories from a common root. [Getting to a single codebase](https://www.wired.com/2015/09/google-2-billion-lines-codeand-one-place/) makes it cleaner to build and push any number of immutable releases across various environments. The best example of violating this is when your app is composed of a dozen or more code repositories. While using one code repository to produce multiple applications can be workable, the goal is a 1:1 relationship between apps and repos. Operating from one codebase can be done but is not without its own challenges. Sometimes one application per repository is the simplest thing that works for a team or organization.

## 2. Dependency Management

Most Java (and Groovy) developers can take advantage of facilities like Maven (and Gradle), which provide the means to declare the dependencies your app requires for proper build and execution. The idea is to allow developers to declare dependencies and let the tool ensure those dependencies are satisfied and packaged into a single binary deployment artifact. Plugins like Maven Shade or [Spring Boot](http://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/) enable you to bundle your application and its dependencies into a single “uberjar” or “fat jar” and thus provide the means to isolate those dependencies.

Figure 1 is a portion of an example Spring Boot application Maven build file, pom.xml. This shows the dependency declarations as specified by the developer.

Figure 1: A portion of POM.xml showing application dependencies

<parent>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-parent</artifactId>

<version>1.3.7.RELEASE</version>

<relativePath/> <!-- lookup parent from repository

</parent>

<dependencies>

<dependency>

<groupId>org.springframework.cloud</groupId>

<artifactId>spring-cloud-starter-config</artifactId>

</dependency>

<dependency>

<groupId>org.springframework.cloud</groupId>

<artifactId>spring-cloud-starter-eureka</artifactId>

</dependency>

<dependency>

<groupId>org.springframework.cloud</groupId>

<artifactId>spring-cloud-starter-zipkin</artifactId>

</dependency>

Figure 2 is a portion of listed dependencies within the same application, showing JARs bundled into the application’s uberjar, which isolates those dependencies from variations in the underlying environment. The application will rely upon these dependencies rather than potentially conflicting libraries present in the deployment target.

Figure 2: A portion of mvn dependency:tree for a sample application

1

[INFO] Scanning for projects...

[INFO]

[INFO] ------------------------------------------------------------------------

[INFO] Building quote-service 0.0.1-SNAPSHOT

[INFO] ------------------------------------------------------------------------

[INFO]

[INFO] --- maven-dependency-plugin:2.10:tree (default-cli) @quote-service ---

[INFO] com.example:quote-service:jar:0.0.1-SNAPSHOT

## 3. Build, Release, Run

A single codebase is taken through a build process to produce a single artifact; then merged with configuration information external to the app. This is then delivered to cloud environments and run. Never change code at runtime! The notion of Build leads naturally to continuous integration (CI), since those systems provide a single location that assemble artifacts in a repeatable way.

Modern Java frameworks can produce uberjars, or the more traditional WAR file, as a single CI-friendly artifact. The Release phase merges externalized configuration (see Configuration below) with your single app artifact and dependencies like the JDK, OS, and Tomcat. The goal is to produce a release that can be executed, versioned, and rolled back. The cloud platform takes the release and handles the Run phase in a strictly separated manner.

## 4. Configuration

This factor is about externalizing the type of configuration that varies between deployment environments (dev, staging, prod). Configuration can be everywhere: littered among an app’s code, in property sources like YAML, Java properties, environment variables (env vars), CLI args, system properties, JNDI, etc. There are various solutions — refactor your code to look for environment variables.

For simpler systems, a straightforward solution is to leverage Java’s System.getenv() to retrieve one or more settings from the environment, or a Map of all keys and values present. Figure 3 is an example of this type of code.

Figure 3: A portion of POM.xml showing application dependencies

**private** String userName = System.getenv(“BACKINGSERVICE\_UID”);  
**private** String password = System.getenv(“BACKINGSERVICE\_PASSWORD”);

For more complex systems, [Spring Cloud](https://cloud.spring.io/spring-cloud-config/spring-cloud-config.html) and [Spring Boot](http://docs.spring.io/spring-boot/docs/current/reference/html/boot-features-profiles.html) are popular choices and provide powerful capabilities for source control and externalization of configuration data.

## 5. Logs

Logs should be treated as event streams: a time-ordered sequence of events emitted from an application. Since you can’t log to a file in a cloud, you log to stdout/stderr and let the cloud provider or related tools handle it. For example, Cloud Foundry’s [loggregator](https://docs.cloudfoundry.org/loggregator/architecture.html" \t "_blank) will turn logs into streams so they can be aggregated and managed centrally. stdout/stderr logging is simple in Java:

1

Logger **log** = Logger.getLogger(MyClass.**class**.getName());  
**log**.setLevel(**Level**.**ALL**);  
ConsoleHandler **handler** = new ConsoleHandler();  
**handler**.setFormatter(new SimpleFormatter());  
**log**.addHandler(**handler**);  
**handler**.setLevel(**Level**.**ALL**);  
**log**.fine(“This **is** fine.”);

## 6. Disposability

If you have processes that take a while to start up or shut down, they should be separated into a backing service and optimized to accelerate performance. A cloud process is disposable — it can be destroyed and created at any time. Designing for this helps to ensure good uptime and allows you to get the benefit of features like auto-scaling.

## 7. Backing Services

A backing service is something external your app depends on, like a database or messaging service. The app should declare that it needs a backing service via an external config, like YAML or even a source-controlled [config server](https://cloud.spring.io/spring-cloud-config/spring-cloud-config.html). A cloud platform handles binding your app to the service, ideally attaching and reattaching without restarting your app. This loose coupling has many advantages, like allowing you to use the [circuit breaker](http://martinfowler.com/bliki/CircuitBreaker.html) pattern to gracefully handle an outage scenario.

## 8. Environmental Parity

Shared development and QA sandboxes have different scale and reliability profiles from production, but you can’t make snowflake environments! Cloud platforms keep multiple app environments consistent and eliminate the pain of debugging environment discrepancies.

## 9. Administrative Processes

These are things like timer jobs, one-off scripts, and other things you might have done using a programming shell. Backing Services and other [capabilities](https://cloud.spring.io/spring-cloud-config/spring-cloud-config.html) from cloud platforms can help run these, and while Java doesn’t (currently) ship with a shell like Python or Ruby, the ecosystem has lots of options to make it easy to run one-off [tasks](http://projects.spring.io/spring-batch/) or make a [shell interface](https://projects.spring.io/spring-shell/).

## 10. Port Binding

In the non-cloud world, it’s typical to see several apps running in the same container, separating each app by port number and then using DNS to provide a friendly name to access. In the cloud you avoid this micromanagement — the cloud provider will manage port assignment along with routing, scaling, etc.

While it is possible to rely upon external mechanisms to provide traffic to your app, these mechanisms vary among containers, machines, and platforms. Port binding provides you full control over how your application receives and responds to requests made of it, regardless of where it is deployed.

## 11. Process

The original 12-factor definition here says that apps must be stateless. But some state needs to be somewhere, of course. Along these lines, this factor advocates moving any long-running state into an external, logical backing service implemented by a cache or data store.

## 12. Concurrency

Cloud platforms are built to scale horizontally. There are design considerations here — your app should be disposable, stateless, and use share-nothing processes. Working with the platform’s process management model is important for leveraging features like auto-scale, blue-green deployment, and more.

## 13. Beyond 12 Factor: Telemetry, Security, API-First Design

The 12 Factors were authored circa 2012. Let’s look at just a few of the many baseline capabilities from modern clouds that make your app more sustainable to run:

* [Health](https://docs.cloudfoundry.org/adminguide/hm-notifications.html) alerts, cloud system [metrics](https://docs.cloudfoundry.org/loggregator/all_metrics.html), [logs](https://docs.cloudfoundry.org/loggregator/architecture.html).
* [Domain-specific](http://docs.spring.io/spring-boot/docs/current-SNAPSHOT/reference/htmlsingle/#production-ready-application-info) telemetry.
* Application performance monitoring (APM).

On Cloud Foundry, Java app logs can simply be directed to stdout/stderr, where they are streamed and aggregated for operators. Spring Boot makes [JMX](http://docs.spring.io/spring-boot/docs/current-SNAPSHOT/reference/htmlsingle/) a snap, and commercial cloud platforms can provide advanced capabilities like APM.

Security external to your application, applied to application endpoints (URLs) with [RBAC](https://en.wikipedia.org/wiki/Role-based_access_control), is important on cloud platforms for SSO & OAUTH2 provider integration. Otherwise, security for multiple Java apps becomes unmanageable.

[Beyond the 12 Factor App](https://pivotal.io/beyond-the-twelve-factor-app) describes the API-first approach as ”an extension of the contract-first development pattern, where developers concentrate on building the edges or seams of their application first. With the integration points tested continuously via CI servers, teams can work on their own services and still maintain reasonable assurance that everything will work together properly.”

## Replatforming

In conclusion, it’s important to realize that you don’t need all 15 factors just to replatform an existing app to run on the cloud. This [cloud-native maturity model](https://twitter.com/wattersjames/status/664044293250641920) (expressed by a large financial services organization) illustrates the type of progression used to approach large, complex monolithic apps and “12 factorize” them incrementally.