12-Factor App Methodology

Factor 1: Codebase

According to the first principle of the 12-Factor App methodology, an application should have **one codebase**, which is stored in a **version control system**. This codebase can be, for example, a Git repository. From this single codebase, deployments can be made to multiple different environments, such as development, test, and production.

The key idea is therefore that **one application = one codebase**, but from this codebase there can be multiple deploys. Deploy here means that the same application is run in different environments or with different configuration.

What does this mean in practice?

If we have a web application, then its entire source code is located in a single repository. From this, for example, the following can be created:

- a **development** deployment for programmers,
- a **staging** system for testing,
- a **production** system for real users.

The three environments are not three separate projects, but three separate runtime instances of the same application.

flowchart TD A[Single Git repository] --> B[Development environment] A --> C[Staging environment] A --> D[Production environment]

Incorrect approach

flowchart TD A[myapp-dev folder] --> X[Development run] B[myapp-test folder] --> Y[Test run] C[myapp-prod folder] --> Z[Production run]

In the first diagram, the three environments start from the same codebase. In the second diagram, there are already three separate copies, which can lead to errors and differences in the long term.

Good approach

- The source code of an application is in one Git repository.
- The repository contains all source files required for the application.
- Development, test, and production deployment all happen from this same repository.

Example:

```
my-webapp/
```

```
  src/
  tests/
  package.json
  Dockerfile
  README.md
```

In this case, `my-webapp` is a single application, and it has a single codebase.

Bad approach

It is not good if the same application has multiple separate, manually synchronized copies, for example:

```
my-webapp-dev/
my-webapp-test/
my-webapp-prod/
```

This is problematic because over time the three versions diverge from each other, and it is no longer possible to know with certainty which one is the current or correct version.

It is also a bad solution if a single repository contains multiple independent applications that actually have separate lifecycles.

Why is this important?

This principle is important because it reduces chaos during development and operations. If an application has multiple “half-identical” codebases, then it can very easily happen that:

- a bug fix is added only to one version,
- the tested version does not match the production version,
- version tracking becomes difficult,
- team members are not working with the same source code.

A single codebase ensures that everyone builds on the same foundation.

Example 1: simple web application

Suppose we create a Python FastAPI-based system. The project repository could be this:

```
invoice-service/
  app/
    main.py
    routes.py
  requirements.txt
  Dockerfile
```

From this same codebase, it can run:

- on the developer's machine on localhost,
- on the test server,
- in the production Docker container.

The difference is not in the source code, but in the configuration and the runtime environment.

Example 2: Node.js application

In the case of a Node.js backend, the same principle applies:

```
student-api/
  src/
  package.json
  package-lock.json
  .env.example
```

The development, staging, and production system are all built from this same repository. We do not create separate `student-api-prod` or `student-api-final` folders.

Example 3: what many people do incorrectly

In beginner projects, it is common for someone to do this:

```
projekt/
projekt_uj/
projekt_vegso/
projekt_vegso_jav/
projekt_vegleges_tenyleg/
```

This is not version control, but manual copying. According to the 12-Factor approach, this should be replaced with a Git repository, where versions can be tracked with commits, branches, and tags.

Relationship with version control

The codebase principle is closely related to version control. The most typical tool for this is Git. Developers use the same repository, and they can work on separate branches there. Even so, the application itself still has a single codebase.

Typical misunderstandings

Many people think that if there are multiple microservices, then that violates this principle. In reality it does not, because in this case each microservice counts as a separate application, so each one can have its own codebase.

For example:

- `user-service` → separate repository
- `order-service` → separate repository
- `payment-service` → separate repository

This is completely correct, because these are separate applications or components.

Factor 2: Dependencies

According to the second principle of the 12-Factor App methodology, an application must **explicitly declare all of its external dependencies**. This means that the application cannot rely on certain libraries or tools already being installed on the system.

Dependency declaration is usually done **with the help of a dependency management system**. This way, the application defines exactly which external packages, libraries, or frameworks it needs.

The goal is that **the application can be built and run in the same way in any environment**.

What does this mean in practice?

Modern applications often use multiple external libraries, such as:

- web framework
- database client
- JSON processing library
- authentication modules

According to the 12-Factor principle, these dependencies **must all appear in the project configuration**.

If someone downloads the project, the required packages must become installable with a single command.

Good approach

Python example:

`requirements.txt`

```
fastapi==0.110
uvicorn==0.29
pydantic==2.6
```

The project structure could be:

```
invoice-service/
  app/
```

```
  requirements.txt
```

Installation:

```
pip install -r requirements.txt
```

This ensures that every developer uses the same packages.

Node.js example

In Node.js, dependencies are listed in the `package.json` file.

```
{
  "name": "student-api",
  "dependencies": {
    "express": "^4.18.2",
    "jsonwebtoken": "^9.0.0"
  }
}
```

Installation:

```
npm install
```

This automatically downloads all required packages.

Java example

In Java projects, Maven or Gradle is commonly used.

Maven example:

```
pom.xml
```

```
<dependencies>
  <dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-web</artifactId>
  </dependency>
</dependencies>
```

Dependencies are downloaded automatically from the Maven repository.

Bad approach

It is incorrect practice if the project implicitly assumes that certain libraries are already installed.

Example:

```
# used in code

import fastapi
import pandas
```

but there is no:

```
requirements.txt
```

In this case, the application will not start on another developer's machine.

Classic problem

Many developers have already encountered the following error:

```
ModuleNotFoundError: No module named 'fastapi'
```

This typically happens because the project's dependencies are not declared.

Dependency isolation

The 12-Factor principle often goes together with **the use of virtual environments**.

Python example:

```
python -m venv venv
source venv/bin/activate
pip install -r requirements.txt
```

This ensures that the project has its own package set.

Containerized example

When using Docker, dependencies appear in the Dockerfile.

```
FROM python:3.11

WORKDIR /app

COPY requirements.txt .
RUN pip install -r requirements.txt

COPY . .
```

In this case, the container contains every required dependency.

Why is this important?

Explicit dependency management ensures that:

- every developer uses the same environment
- the application can be reproduced at any time
- the build process can be automated
- the CI/CD pipeline works reliably

This is especially important in large teams.

Common misunderstanding

Declaring dependencies does not mean only program libraries.

It may also include, for example:

- build tools
- CLI helper programs
- compilers
- runtime environments

Correct operation

flowchart LR A[Git repository] --> B[Dependency file] B --> C[Automatic installation] C --> D[Run application]

Factor 3: Config

According to the third principle of the 12-Factor App methodology, the application's **configuration must be separated from the source code**. Configuration should not be stored in the program code, but in **environment variables**.

Configuration means values that **can vary by environment**, for example:

- database connection
- API keys
- passwords
- URLs of external services
- debug settings

According to the 12-Factor principle, these **must not be hardcoded in the code**.

What does this mean in practice?

If an application runs in multiple environments (for example development, test, production), then the configuration can be different.

For example:

Environment	Database
development	localhost
test	test-db
production	prod-db

If the database address is in the code, then the program would need to be modified before every deploy. According to the 12-Factor approach, configuration **enters the program as an external parameter**.

Bad approach (hardcoded configuration)

Python example:

```
DATABASE_URL = "postgres://user:password@prod-db:5432/app"
```

In this case:

- the password is in the code
- the configuration is committed into the repository
- the code must be modified when switching environments

This causes security and operational problems.

Good approach: environment variable

Python example:

```
import os

DATABASE_URL = os.getenv("DATABASE_URL")
```

In the environment:

```
export DATABASE_URL=postgres://user:password@prod-db:5432/app
```

This way the code remains the same in every environment.

Node.js example

```
const dbUrl = process.env.DATABASE_URL;
```

Startup:

```
DATABASE_URL=postgres://localhost/mydb node server.js
```

Docker example

In the case of a Docker container, configuration also appears as an environment variable.

```
docker run -e DATABASE_URL=postgres://db/app myapp
```

Kubernetes example

In Kubernetes, configuration often appears in the form of a **ConfigMap** or **Secret**.

```
env:
- name: DATABASE_URL
  valueFrom:
    secretKeyRef:
      name: database-secret
      key: url
```

What counts as configuration?

Typical configuration items:

- database connections
- external API keys
- SMTP server
- cache server
- feature flags
- log levels

What is **not configuration** is the operating logic of the program.

Classic mistake

In many projects you can find a file like this:

```
config.py
```

```
DATABASE_URL = "postgres://localhost/app"
SECRET_KEY = "123456"
```

If this is committed, then:

- passwords may end up in a public repository
- a separate config file is needed for every environment
- the chance of error increases

Advantages of environment variables

The use of environment variables provides several advantages:

- they do not get into the source code
- they can be modified easily during deploy
- safer handling
- works well in containerized systems

Mermaid diagram: configuration handling

flowchart TD A[Application code] --> B[Environment variable] B --> C[Load configuration] C --> D[Application runtime]

Factor 4: Backing Services

According to the fourth principle of the 12-Factor App methodology, the infrastructure components used by the application must be treated as **external services (backing services)**. These are resources that **are not part of the application**, but separate systems that the application connects to over the network.

Typical backing service examples:

- database (PostgreSQL, MySQL)
- cache system (Redis, Memcached)
- message queue (Kafka, RabbitMQ)
- search engine (Elasticsearch)
- object storage (S3)

The important principle is that these services should appear to the application as **replaceable resources**.

What does this mean in practice?

The application logic does not contain the infrastructure itself. The database, cache, or other components run as separate services, and the application only connects to them.

For example, a web application can use a database and a cache system:

flowchart TD A[Application] --> B[Database] A --> C[Cache]

In this model, the database and the cache are separate services.

Replaceability

One important consequence of the backing service principle is that the service is **easy to replace**.

For example, an application can use:

- a local PostgreSQL database during development,
- a cloud-based database in production.

The operation of the application does not change because the database appears as an external service.

Example architecture

flowchart TD A[Web application] A --> B[(PostgreSQL)] A --> C[(Redis)] A --> D[(Message Queue)]

The application treats every external resource as a separate service.

Factor 5: Build, Release, Run

The fifth factor prescribes that the application lifecycle must be divided into **three well-separated phases**:

- **Build**
- **Release**
- **Run**

Separating the three phases is important because this makes deployment **reproducible, automatable, and safe**.

Meaning of the three phases

Build

During the build phase, a **runnable package (artifact)** is created from the source code.

In this step, for example, the following happens:

- source code compilation
- dependency installation
- creation of the package or container image

Examples:

- Java → JAR file
- Node.js → built application
- Docker → container image

Release

The release phase is the **combination of the build result and the configuration**.

At this point, a concrete application version is created that can be deployed.

So the release is:

build + configuration

Run

During the run phase, the application **is executed in the selected environment**.

This can be, for example:

- server
- container
- cloud platform

The run phase **does not modify the build anymore**, it only starts the application.

Relationship of the three phases

flowchart LR A[Source Code] --> B[Build] B --> C[Release] C --> D[Run]

Why is the separation important?

If the three phases are not separated from each other, then it will be difficult to:

- reproduce the deployment
- roll back to an earlier version
- automate the deploy process

The Build-Release-Run model ensures that **the same build can be run in multiple environments**.

Example in a Docker environment

Build:

```
docker build -t myapp:1.0 .
```

Release:

```
docker tag myapp:1.0 registry/myapp:1.0
```

Run:

```
docker run myapp:1.0
```

Example in a CI/CD pipeline

A modern pipeline often follows exactly these three steps.

flowchart LR A[Git commit] --> B[CI build] B --> C[Release artifact] C --> D[Deployment]

Factor 6: Processes

According to the sixth factor, the application should consist of **stateless processes**. This means that running instances of the application must not store persistent state in their own memory or local filesystem.

Persistent data must always be stored in an **external service**, for example in a database or cache system.

What does stateless operation mean?

Stateless means that any running instance can handle a request without knowing the state of previous requests.

If the application runs in multiple instances, then the system can route requests to any of the instances.

Example of a stateless architecture

flowchart TD A[Load Balancer] A --> B[App instance 1] A --> C[App instance 2] A --> D[App instance 3] B --> E[(Database)] C --> E D --> E

In this model, multiple application instances run at the same time. Each of them connects to the same database, and none of them stores persistent data in itself.

Bad approach: state in memory

It is an incorrect solution if the application stores user state in its own memory.

Example:

```
sessions = {}

def login(user):
    sessions[user.id] = "active"
```

If the application runs in multiple instances, then the state in the memory of one instance will not be available to the other instance.

Good approach: state in an external service

User state is stored in an external system.

For example, in Redis:

flowchart TD A[App instance 1] --> B[(Redis)] C[App instance 2] --> B

This way every application instance sees the same state.

File handling

The stateless principle also applies to files.

Incorrect solution:

```
/tmp/uploads/file1.jpg
```

If the application is moved to a new instance, the file may disappear.

Good solution:

- object storage (S3)
- network filesystem
- database

Why is this important?

Stateless processes make the following possible:

- horizontal scaling
- containerized deploy
- automatic restart

- operation of load balancing

In a modern cloud environment

Cloud systems often automatically start and stop application instances.

If the application is stateless, then this does not cause a problem.

Factor 7: Port Binding

The seventh factor states that the application **publishes itself as a standalone service through a port**. The application starts its own HTTP or network server, and becomes available through it.

This means that the application **does not depend on an external web server**, but provides service access itself.

What does this mean in practice?

In many older systems, the application connects to an external web server.

Example:

- Apache
- Nginx
- IIS

In this model, the web server loads the application, for example with the help of a plugin or module. In contrast, the 12-Factor approach suggests that the application should **start its own server** and be available through a port.

Example for a modern web application

A Node.js or Python web application often starts like this:

```
app.listen(8080)
```

or

```
uvicorn main:app --port 8000
```

The application then becomes directly available on the specified port.

Architecture example

flowchart TD A[Client] --> B[Application server :8000]

The application runs its own server and directly accepts requests.

Containerized environment

In containerized systems, this is especially important.

A Docker container typically serves on one port.

Example:

```
docker run -p 8000:8000 myapp
```

The application runs on port 8000 inside the container.

Microservice architecture

In microservice systems, every service publishes itself on its own port.

flowchart TD A[API Gateway] --> B[User Service :8001] A --> C[Order Service :8002] A --> D[Payment Service :8003]

Every service is available on a separate port.

Why is this important?

Port binding enables:

- simple service startup
- containerized execution
- microservice architecture
- dynamic infrastructure

The platform (for example Kubernetes or a cloud provider) can automatically connect the services.

Factor 8: Concurrency

The eighth factor describes how the application can be **scaled**. According to the 12-Factor App methodology, the application can be **scaled by starting multiple processes**, rather than by

strengthening a single larger process. This means that if the load increases, then **multiple identical application instances** are started in parallel.

What does this mean in practice?

There are two basic ways to scale:

- **vertical scaling** – using a more powerful server
- **horizontal scaling** – starting more instances

The 12-Factor approach supports **horizontal scaling**.

Example of horizontal scaling

flowchart TD A[Load Balancer] A --> B[App instance 1] A --> C[App instance 2] A --> D[App instance 3]

The load balancer distributes requests among multiple running instances.

Process types

Applications often contain several different types of processes.

For example:

- web server
- background processor (worker)
- scheduled task (scheduler)

These can be run as separate processes.

Example architecture

flowchart TD A[Client] --> B[Web process] B --> C[(Database)] B --> D[Worker process]

The web process accepts requests, while the worker performs background tasks.

Example of worker scaling

If many background tasks arrive, more workers can be started.

flowchart TD A[Queue] --> B[Worker 1] A --> C[Worker 2] A --> D[Worker 3]

In a containerized environment

In modern cloud systems, scaling is often automatic.

Example in Kubernetes:

- 1 pod → low load
- 5 pods → high load

The system can automatically start new instances.

Why is this important?

Using multiple processes makes the following possible:

- handling higher load
- better fault handling
- flexible scaling
- use of cloud infrastructure

Factor 9: Disposability

According to the ninth factor, the application's processes should **start quickly and stop quickly**. Such processes can be created or terminated easily without interrupting the operation of the system.

This property is especially important in modern cloud and containerized environments, where application instances often start and stop automatically.

What does this mean in practice?

If a new instance of the application needs to be started (for example because of increased load), then it should **start within a few seconds**.

If an instance has to be stopped (for example because of an update or scaling), then the application should **safely finish its running operations**, and then stop.

Fast startup

Fast startup makes it possible for the system to start new instances when the load increases.

flowchart LR A[Load increase] --> B[Start new application instance] B --> C[New instance accepts requests]

Fast shutdown

When stopping a process, the application must be given a chance to finish running operations.

Example process:

1. the system signals shutdown, 2. the application finishes the current requests, 3. the process exits.

Example with a worker process

A background processor (worker) can finish its current task before shutdown.

flowchart LR A[Stop signal] --> B[Process finishes the current task] B --> C[Process stops]

Why is this important?

Fast startup and shutdown make the following possible:

- automatic scaling
- fast deploy process
- replacement of faulty instances
- containerized execution

Modern cloud systems often start new instances in a short time.

Example in a containerized environment

For example, a Kubernetes system can start a new container if the load increases. If one instance is faulty, the system stops it and starts a new one.

Factor 10: Dev/Prod Parity

The tenth factor states that the difference between the **development**, **staging**, and **production** environments should be as small as possible.

The more similar the environments are to each other, the smaller the chance that the application works during development but causes an error in the production system.

The classic problem

In many systems, the development and production environments are very different.

For example:

Development	Production
SQLite database	PostgreSQL
local filesystem	cloud storage
simple server	system made up of multiple servers

In such cases, it often happens that the program works during development, but causes an error in production.

Correct approach

The goal is that the difference between environments should be **minimal**.

Ideally:

- the same database type
- the same runtime environment
- the same infrastructure

Factor 11: Logs

According to the eleventh factor, the application **does not manage log files directly**, but writes logs as a **stream** to standard output.

The collection, storage, and processing of logs is done **by the runtime environment**, not by the application itself.

What does this mean in practice?

The application simply writes log messages to standard output.

For example:

```
print("User logged in")
```

or

```
console.log("Server started")
```

The application does not create its own log files and does not deal with archiving logs.

Traditional approach

Older systems often write logs directly to a file.

Example:

```
/var/log/myapp.log
```

This can cause multiple problems:

- the log file size continuously grows
- it is difficult to collect logs from multiple servers
- log handling becomes the application's responsibility

Correct approach

The application only outputs events, and the platform collects the logs.

flowchart LR A[Application] --> B[Stdout log stream] B --> C[Log collection system]

The log collection system can be, for example:

- log aggregator
- monitoring system
- cloud log service

Example in a containerized environment

In the case of Docker, logs automatically go to standard output.

The Docker system collects and stores them.

flowchart LR A[Application container] --> B[stdout] B --> C[Docker log system]

In a cloud environment

In cloud systems, logs often go into a central system.

For example:

- ELK stack
- Loki
- cloud logging service

flowchart LR A[Application] --> B[Log stream] B --> C[Central log storage]

Factor 12: Admin Processes

According to the twelfth factor, the **administrative or maintenance tasks** belonging to the

application must be run as separate processes, in the same environment as the application itself. Administrative tasks are not part of the normal application process, but **one-time or periodic operations** that must be started separately.

Examples of administrative processes

Typical administrative operations:

- database migration
- running a data-fix script
- clearing the cache
- importing data
- one-time maintenance task

These tasks do not run as part of the web application, but as separately started processes.

Example: database migration

Before updating an application, it is often necessary to modify the database structure.

```
python manage.py migrate
```

This is an administrative process that must be started separately.

Example architecture

flowchart TD A[Web Application] B[Admin Process] A --> C[(Database)] B --> C

The admin process uses the same database as the application, but runs separately.

Important rule

Admin processes must **use the same environment** as the application.

This means:

- the same codebase
- the same configuration
- the same dependencies

This ensures that admin operations work in the same way as the application.

Example in a containerized environment

In a containerized system, an admin task can also be run as a separate container.

flowchart TD A[Application container] --> C[(Database)] B[Migration container] --> C

The migration container runs only once.

From:
https://edu.iit.uni-miskolc.hu/ - **Institute of Information Science - University of Miskolc**

Permanent link:
https://edu.iit.uni-miskolc.hu/tanszek:oktatas:iss_t:12-factors

Last update: **2026/03/30 18:08**