The Evolutionary Architect

Contents

[An Evolutionary Vision for the Architect 2](#_Toc57731687)

[Zoning 2](#_Toc57731688)

[A Principled Approach 3](#_Toc57731689)

[Strategic Goals 3](#_Toc57731690)

[Principles 3](#_Toc57731691)

[Heroku’s 12 Factors App 3](#_Toc57731692)

[Practices 7](#_Toc57731693)

[Combining Principles and Practices 7](#_Toc57731694)

[A Real-World Example 7](#_Toc57731695)

[The Required Standard 8](#_Toc57731696)

[Monitoring 8](#_Toc57731697)

[Interfaces 8](#_Toc57731698)

[Architectural Safety 8](#_Toc57731699)

[Governance Through Code 8](#_Toc57731700)

[Exemplars 8](#_Toc57731701)

[Tailored Service Template 8](#_Toc57731702)

[Technical Debt 8](#_Toc57731703)

[Exception Handling 8](#_Toc57731704)

[Governance and Leading from the Centre 8](#_Toc57731705)

[Building A Team 8](#_Toc57731706)

# An Evolutionary Vision for the Architect

The software architects should behave like architects, or rather than town planners. Everything cannot be foreseen, and so rather than planning for any eventuality, we should plan to allow everything that will happen, and so rather than plan for any eventuality, we should plan to allow for change by avoiding the urge to overspecifiy every last thing.

# Zoning

What are our zones ??

These are our service boundaries, or perhaps coarse-grained group of services.

As architects, we need to worry more about what happens between the zones than what happens inside the zones. *We need to think more about how our services talk to each other, or ensuring that we can properly monitor the overall health of our system.*

Within each service, you may be OK with the team who owns that zone picking a different technology stack or data store.   
Concerns here could be :

1. It becomes harder to hire people or them between teams if you have 10 different technology stacks to support.
2. If each team picks a completely different data store, you may find yourself lacking enough experience to run any of them at scale.

Between services, things could get messy.

For example, if one service decides to expose REST over HTTP, another makes use of protocol buffers, a third uses Java RMI, then integration can become a nightmare as consuming service have to understand and support multiple styles of interchange.

“Be worried about what happens between the boxes, and be liberal in what happens inside.”

# A Principled Approach

“Rules are for the obedience of fools and the guidance of wise men.”

## Strategic Goals

Strategic goals should speak to where your company is going, and how it sees itself as best making its customers happy. These will be high-level goals and may not include technology at all. They could be defined at a company level or a division level.

## Principles

Principles are rules you have made in order to align what you are doing to some larger goal, and will sometimes change.

You probably don’t want loads of these. Fewer than 10 is a good number – small enough that people can remember them, or to fit on small posters.

The more principles you have, the greater the chance that they overlap or contradict each other.

### Heroku’s 12 Factors App

#### Who Should Use Heroku’s 12-factor app ??

The 12-factor app is a methodology for building software-as-a-service apps that:

* Use declarative formats for setup automation, to minimize time and cost for new developers joining the project;
* Have a clean contract with underlying OS, offering maximum portability between execution environments;
* Are suitable for deployment on modern cloud platforms, obviating the need for servers and systems administration;
* Minimize divergence between development and production, enabling continuous deployment for maximum agility;
* And can scale up without significant changes to tooling, architecture or development practices.

The 12-factor methodology can be applied to apps written in any language, and which use any combination of backing services.

#### The Twelve Factors

[The Twelve-Factor App (12factor.net)](https://www.12factor.net/)

##### Codebase

(One codebase tracked in revision control, many deploys)

A 12-factor app is always tracked in version control system, such as Git, Mercurial or Subversion. A copy of the revision tracking database is known as a code repository, often shortened to *code repo* or just *repo*.

A *codebase* is any single repo, or any set of repos who share a root commit.

There is always a one-to-one correlation between the codebase and the app:

1. If there are multiple codebases, it’s not an app – it’s a distributed system. *Each component is a distributed system in an app. And each can individually comply with twelve-factor.*
2. 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.*

##### Dependencies

(Explicitly declare and isolate dependencies/*bundling*/*vendoring*)

A twelve-factor app never relies on implicit existence of system-wide packages. It declares all dependencies, completely and exactly, via a *dependency declaration* manifest. Also, it uses a *dependency isolation* tool during execution to ensure that no implicit dependencies “leak in” from the surrounding system.

The full and explicit dependency specification is applied uniformly to both production and development.

|  |  |  |
| --- | --- | --- |
| Language | Dependency Declaration | Dependency Isolation |
| Ruby | Gemfile | bundle exec |
| Python | Pip | virtualenv |
| C | Autoconf | static linking |

##### Config

(Store config in the environment)

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.

“A litmus test for whether an app has all config correctly factored out of the code is whether the codebase could be made open source at any moment, without compromising any credentials.”

**The twelve-factor app stores config in environment variables.**

Another aspect of config management is grouping. Sometimes apps batch config into named groups named after specific deploys, such as *development*, *test* and *production* environments.

##### 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,

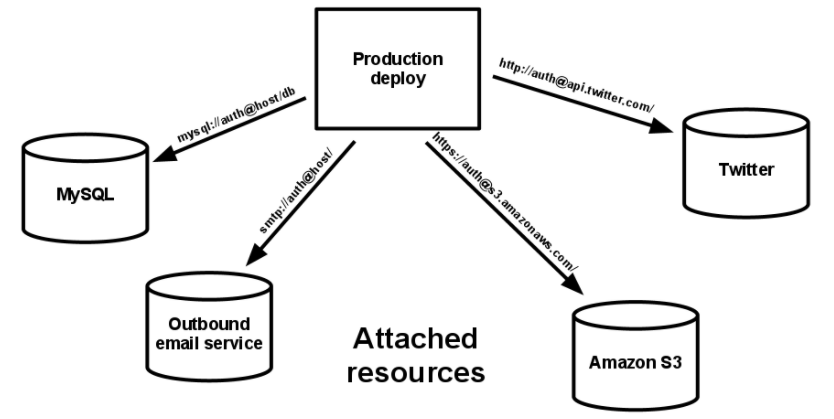
1. Datastores (such as MySQL or CouchDB)
2. Messaging/queueing systems (such as RabbitMQ or Beanstalkd)
3. SMTP services for outbound email (such as Postfix)
4. Caching systems (such as Memcached)

Backing services like the databases are traditionally managed by the same systems administrators who deploy the app’s runtime.

Examples include,

1. SMTP services (such as Postmark)
2. Metrics-gathering services (such as New Relic or Loggly)
3. Binary Asset services (such as Amazon S3)
4. API-accessible consumer services (such as Twitter, Google Maps or Last.fm)

**The code for a twelve-factor app makes no distinction between local and third-party services. To the app, both are attached resources, accessed via URL or local/credentials stored in the config.**

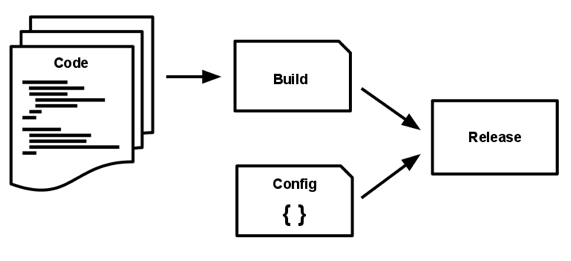
A deploy of the twelve-factor app should be able to swap out a local MySQL database with one managed by a third-party without any changes to the app’s code. Only the resource handle in the config needs to change.

##### Build, Release, Run

(Strictly separate build and run stages)

**The 12-factor app uses strict separation between the build, release and run stages.**

A 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* andcompiles binaries and assets.
* The *release stage* takes the build produced by the build stage and combines it with the deploy’s current *config*. The resulting *release* contains both the build and the config and is ready for immediate execution in the execution environment.
* The *run* stage runs the app in the execution environment, by launching some set of the app’s *processes* against a selected release.

Deployment tools typically offer release management tools, most notably the ability to roll back to a previous release.

Every release should always have a unique release ID. Builds are initiated by the app’s developers whenever new code is deployed.

Runtime execution, by contrast, can happen automatically in case such as server reboot, or a cras

##### Processes

(Execute the app as one or more stateless processes)

##### Port Binding

(Export services via port binding)

##### Concurrency

(Scale out via the process model)

##### Disposability

(Maximize robustness with fast startup and graceful shutdown)

##### Dev/Prod Parity

(Keep development, staging and production as similar as possible)

##### Logs

(Treat logs as event streams)

##### Admin Process

(Run admin/management tasks as one-off processes)

## Practices

## Combining Principles and Practices

## A Real-World Example

# The Required Standard

## Monitoring

## Interfaces

## Architectural Safety

# Governance Through Code

## Exemplars

## Tailored Service Template

# Technical Debt

# Exception Handling

# Governance and Leading from the Centre

# Building A Team