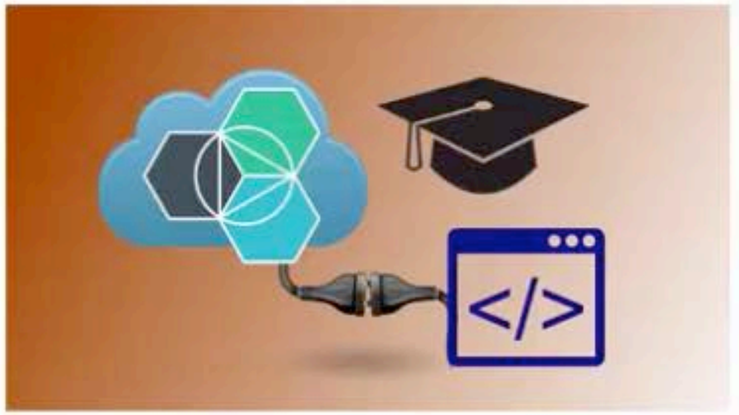
Intro - 12 Factor App



Learning Objectives:

1. Traditional versus PaaS infrastructure

2. What 12 factor app stands for?

Traditional versus PaaS

Traditional	PaaS
Full control over physical resources (VM)	**Limited control** over physical resources (VM)
Uses the scale **up/out** strategy	Prefers the scale **out** strategy; Auto scale
Local storage is **persistent**	Local storage is **ephemeral**
Data tier is co-located with compute	Data tier abstracted & distributed
Manual processes to handle failure	**Automations** handle failure (self healing)
Fixed costs for the resource tier (capex)	**Flexible** costs controlled by the apps (opex)

Cloud Native Application

- Traditional architecture and design practices for application are not aligned with the cloud platform
- Cloud native applications:
  - Built to be self healing (automation & redundancy)
  - Take advantage of the cloud computing platform(s)
  - Scale up or down based on the defined policies
  - Designed for failure

12 factor methodology

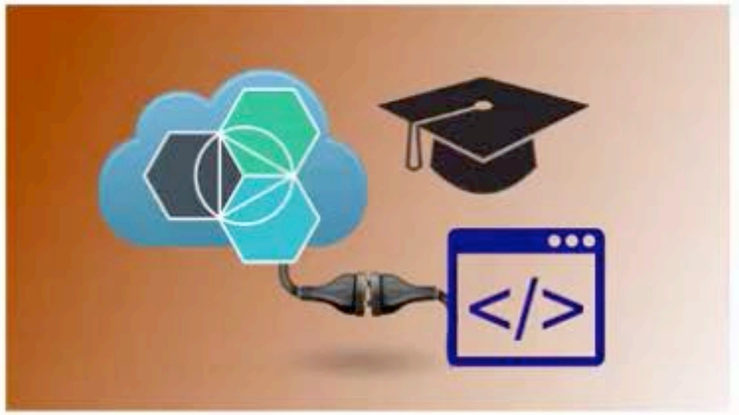
http://12factor.net/

- Best practices for the development of applications meant to be deployed on a cloud platform
- The 12 factor app is a methodology for building SaaS that:
  - Uses declarative format for setup automation
  - Suitable for deployment on the cloud platform
  - Clean contract with the underlying resources to maximize portability
  - Minimize divergence between production and development environments

Summary

- Traditional application design patterns are not aligned with the apps developed for the cloud platform
- The 12 factor app refers to applications that follow the 12 best practices put together for the cloud applications

12 Factor App #1 - 4



Learning Objectives:

1. Codebase
2. Dependencies
3. Configuration
4. Backing Services

THE TWELVE-FACTOR APP

#1 Codebase

THE TWELVE-FACTOR APP

"One codebase tracked in revision control, many deploys"

- Single codebase for apps in revision control (GIT, Subversion…)
- Each app in its own repository; Branches used for deployments to environments

#2 Dependencies

"Explicitly declare and manage dependencies"

- Explicitly declare all app dependencies such as Jar files and node JS packages
- Automate the build process – repeatable deployments

- E.g., node Js applications list dependencies in *package.json*
  npm command takes care of downloading & packaging the dependencies

- E.g., use Maven for Java/Spring apps.
  1. explicitly declares the dependencies in Maven files
  2 maven builds app by pulling and packaging dependencies in app war file

#3 Configuration

"Store configuration in the environment"

- Anything that changes from environment to environment
- Do not place configuration information in the code or property files
- Use environment variables for storing config information
    - E.g., User defined environment variables may be used by developer for setting application specific configuration

#4 Backing services

"Treat backing services as attached services"

- Attached service = App refers to the service by way of a URL that is provided via environment variables.
- Attach using *cf bind* ; preferably by using manifest file
- Swapping the service would not require any code change
    - E.g., Use user defined service to expose an external data source as a *service* for which url is provided via the environment variable

Summary

- Codebase – manage all code in revision control system. Same codebase for app deployed to multiple environments.
- Dependencies – should be managed explicitly. No dependency should be implicit.
- Configuration – manage configuration in environment variable. Do not store config information in codebase or property files.
- Backing service – app refers to external services by way of URL; app exposes itself by way of URL so that it can be used as a backing service