**SOA 2.0**

**12 Factors App**

**Guidelines for Cloud Native Apps**

Document Owner : SOA CoE, HKIT

Last Review : 3rd, October, 2017

Version : 0.2

**Version History**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Version** | **Date (DD/MM/YYYY)** | **Description of Change** | **Reviewed By** | **Approved By** |
| 0.1 | 03/10/2017 | Initial Draft | Chandi Prasad Ojha | Angus Chan |
| 0.2 | 27/11/2017 | Added 3 new Factors | Chandi Prasad Ojha |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |

**Table of Contents**

[1 Introduction 3](#_Toc500325666)

[1.1 Purpose 3](#_Toc500325667)

[1.2 12 Factor App Summary View 4](#_Toc500325668)

[1.3 12 Factor Apps Value Proposition 4](#_Toc500325669)

[1.4 Acronyms & Definitions 5](#_Toc500325670)

[2 Application Codebase 6](#_Toc500325671)

[3 Application Dependency Management 7](#_Toc500325672)

[4 Build, Release & Run 8](#_Toc500325673)

[5 Configurations & Credentials Management 9](#_Toc500325674)

[6 Backing Services 10](#_Toc500325675)

[7 Logs 11](#_Toc500325676)

[8 Dev/Prod Parity 12](#_Toc500325677)

[9 Disposability - Rapid Start and Graceful Shutdown 13](#_Toc500325678)

[10 Admin / Batch Tasks 14](#_Toc500325679)

[11 Port Binding 15](#_Toc500325680)

[12 Processes 16](#_Toc500325681)

[13 Concurrency 17](#_Toc500325682)

[14 Beyond 12 Factors – API First 18](#_Toc500325683)

[15 Beyond 12 Factors – Telemetry (Monitoring) 19](#_Toc500325684)

[16 Beyond 12 Factors – Security First Design 20](#_Toc500325685)

[17 Appendix 21](#_Toc500325686)

# Introduction

## Purpose

12 Factor Apps is a set of guidelines published in 2012 by a team at HEROKU. The objective of these 12 factors is to teach developers how to build cloud ready applications using declarative formats for automated setups, had clean contract with underlying operating system and were able to scale dynamically.

The 12 Factor App guidelines help achieve the following -

1. Build cloud native applications and services
2. Developers become productive quickly
3. Enable more frequent deliveries following DevOps based delivery model.
4. Scale applications significantly without changing design OR architecture

This document provides a brief overview about the 12 Factor App and its principles. It would help the developers working on micro services to familiarise themselves with 12 Factor Apps explaining what this 12 Factor App is all about and how it helps

Each of these factors have been described as an exclusive topic containing following sections -

1. Overview – Brief overview about the factor
2. Detailed Description – Explaining benefits, violation scenario’s and key takeaways
3. PCF Context – Provides pointers on how PCF platform enables / supports this factor

**NOTE -** On top of these 12 factors, Pivotal has come-up with 3 new factors which will further improve the value proposition of cloud native application development.

## 12 Factor App Summary View

|  |  |
| --- | --- |
| **12 Factors** | **Summary Description** |
| Codebase | One codebase tracked in revision control. Many Deploys |
| Dependencies | Explicitly declare and isolate dependencies |
| Configurations | Externalize application configurations and store them in the environment |
| Backing Services | Consider backing services as attached resources |
| Build, Release, Run | Build, Release and Run stages should be strictly separated and mutually exclusive |
| Processes | Execute the application as one or more stateless processes |
| Port Binding | Export services via port binding |
| Concurrency | Scale out via the process model |
| Disposability | Maximize robustness by fast start up and graceful shutdown |
| Dev / Prod Parity | Keep Development, Staging and Production regions as similar as possible |
| Logs | Consider logs as event streams |
| Administrative Processes | Run admin processes & management tasks as one-off processes |
| **Beyond 12 Factors** | **Summary Description** |
| API Frist Design | Follow API First Design approach while designing applications |
| Telemetry | Apps deployment over cloud should be considered as Space Probes |
| Security First Design | Security should never be an afterthought. |

## 12 Factor Apps Value Proposition

|  |  |  |
| --- | --- | --- |
| **S No** | **12 Factors** | **Value Proposition** |
| 1 | One Codebase One App | Time to Market – use good SDLC practices for quick delivery & deployment |
| 2 | Dependency Management | Dev Productivity – Standardize and simplify to avoid last minute surprises |
| 3 | Configurations | Release Management best practices – Move configs to environment VARS |
| 4 | Backing Services | Resilience / Agility – maintain loose binding |
| 5 | Build, Release, Run | Release Management best practices – make use of CI / CD automation |
| 6 | Stateless Processes | Cloud Platform Compatibility – Move state information to backing services |
| 7 | Port Binding | Operations Efficiency – Use PCF features like routing, scaling etc. |
| 8 | Concurrency | Auto-scaling – Design applications for cloud platforms. Use PCF features |
| 9 | Disposability | Auto-scaling – Move slow executing processes to backing services |
| 10 | Dev / Prod Parity | Reliability – use PCF (Cloud platform) offered environments |
| 11 | Logs | Real-time Metrics – use PCF features (STDOUT / STDERR) |
| 12 | Administrative Processes | Reliability – Move admin processes to backing services |
|  |  |  |

## Acronyms & Definitions

|  |  |
| --- | --- |
| **Term** | **Definition** |
| API | Application Programming Interface |
| CLI | Command Line Interface |
| IDE | Integrated Development Environment |
| SDLC | Software Development Life Cycle |
| STS | Spring Tool Suite |
| UR | User Requirements |
| URI | Uniform Resource Identifier |
| URL | Uniform Resource Locator |
|  |  |
|  |  |

# Application Codebase

|  |
| --- |
| **Overview**  A microservice application should have its own exclusive codebase managed using a source code repository. The goal is to have a 1:1 relationship between applications and repositories. |
| **Description**  Maintaining a single codebase per application makes it easy to   1. Automate the build and deployment process 2. Perform any number of releases across various environments (DEV, QA, PROD etc.).   Even though the code base is same, there could be different versions active in each environment. Meaning the developer would be implementing a feature for next release (NR2) which is yet to be committed, while in Test environment release (NR1) is functional and performance testing is in progress and in Production Environment we might be have the version (NR0). All these are possible with the same code base with proper branching strategy and application release processes  In case we have multiple codebase and more repositories,   1. It becomes difficult to automate the build and deployment which affects the application release cycles 2. Here arises a need to analyze the system properly, identify the potential monolith structures within it, decompose the system and split it into multiple micro services   Also there we could see some occurrences of one code base used to deliver multiple applications. In JEE, we can have multiple WAR files representing multiple applications packaged into one single EAR file. This results in violation of the *One Application One Codebase* principle. Analyze these structures to decompose and split into several microservices  **Key takeaways:**   1. We need to ensure each of our services have their own independent repository 2. Analyze the system to ensure no monolith sort of structures exist 3. Have a proper branching strategy 4. Streamline application release process and application version conventions |
| **PCF Context**   1. Majority version control systems have integration with Cloud Foundry 2. PCF provides “Org” and “Space” for deploying and managing different environments (Sandbox, Dev, Test and Prod etc.) |

# Application Dependency Management

|  |
| --- |
| **Overview**  A microservice application should declare all its dependencies through declaration mechanism and leave it to the build system to get those dependencies during the application packaging phase. It shall not have all its dependency as part of the source. |
| **Detailed Description**  For Java, maven and gradle are the most popular tools which provide the means to isolate the application dependency through declaration mechanism. Also we have plugins like Maven shade, Spring Boot etc. which helps to bundle the application and the dependencies into a single JAR/WAR.  During the application build process, the tool takes the responsibility to ensure the dependencies are satisfied.  The major advantage of declaring dependencies is   1. It’s easy to setup the application for the new developers. The developer can check out the application codebase in their system with only the language runtimes and dependency management application (Maven / Gradle) installed as prerequisites. 2. Helps avoid any potential conflicting libraries introducing issues in other environments 3. Make it easy to manage application dependencies   . **Key takeaways:**   1. Always use Maven to manage the application dependencies 2. Use manifest YML to declare the service dependencies and utilize the Java build pack 3. Do not rely on class path mechanism as generally practiced earlier 4. Use the appropriate spring starter templates and spring cloud versions 5. Come up with our own maven project starter templates for different scenario’s to ensure the application uses the recommended and standardized templates of the project |
| **PCF Context**   1. Application runtime, services and framework dependencies can be declared in application manifest YML file 2. CF build pack examines the YML to download dependencies and configure the bounded services |

# Build, Release & Run

|  |
| --- |
| **Overview**  A microservice application should follow strict separation between the build, release, and run stages. It means we would not be able to make changes to the code at runtime, since it would not be possible to propagate the changes back into build stage. |
| **Detailed Description**  The main concept here is not to change code at runtime. The codebase goes through the build process to produce an immutable artifact. This artifact gets merged with configuration information external to the application and finally delivered to cloud environments. The goal is to produce a release that can be executed, versioned, and rolled back. The cloud platform takes the release artifact and handles the Run phase in a strictly separated manner.  Here during the design phase, we do the application design with respect to that particular release. This covers the application dependencies and other aspects from release perspective. The development and unit testing is carried out by the developer using the IDE and other support tools. The build phase is where the code repository is made a versioned binary artifact. Builds are handled by Continuous Integration Servers which can be released and deployed any number of times and different environments  The run phase is typically handled by the platform which then invokes processes to launch the application. Once the application is running, now it’s up to the platform to keep it alive, monitoring the health of the services, aggregating and push logs, scaling the instances etc.  **Key takeaways:**   1. Ensure single code base and declare dependencies 2. Follow proper release mechanism and release versioning strategy 3. The release should be unique 4. Once a release artifact is created, it cannot be modified. Instead we should create a new release in case of any changes required |
| **PCF Context**   1. Code base gets transformed into deployment artifact in 3 stages (Build, Release and Run). 2. PCF provides out of the box support for Concourse Continuous Integration Server. 3. With well provided support for CI, the builds and releases are immutable and any change to code creates a new build and gets it released |

# Configurations & Credentials Management

|  |
| --- |
| **Overview**  A microservice application should externalize the configuration parameters and store them in the environment as they can vary across deployment environments (Dev, Test, Prod etc.). This helps to create immutable artifacts which can be deployed to different environments automatically with CI/CD pipelines |
| **Detailed Description**  All application has some parameters that are very likely to vary between deployments (Local, Dev, Test, Prod etc.). Usually these are done at different places (property sources, environment variables, passed as arguments during application start up, JNDI etc.).  This includes:   * Configurations and credentials related to database, Cache, and other backing services * Certificates and Credentials to external services * Application runtime configurations   Also sometimes the applications store configurations as constants in the code itself. This results in violation of the 12 factor that requires strict separation of configurations from code.  When it comes to cloud native solutions the following are the best options for managing configurations   1. Utilizing configuration server (Spring Cloud Config) backed by Git repository 2. Utilizing environment variables   **Key takeaways:**   1. Do not store configurations as constants within the code 2. Utilize configuration server and environment variables to store configuration related parameters. Come up with recommendations on what parameters to part of Config server / Env Vars 3. Update the CI/CD pipelines to fully automate the build and deployment based on the configuration related recommendations |
| **PCF Context**  CF provides different variables to configure runtime information   1. VCAP\_APPLICATION: App specific attributes 2. VCAP\_SERVICES: Services bound to the application 3. CF\_INSTANCE\_\*: Instance specific variables 4. Environment vars: System wide variables impacts all instances 5. User Provided Service Instances to manage external service credentials (CF CUPS) |

# Backing Services

|  |
| --- |
| **Overview**  A microservice application should leverage backing services for external resource requirements like DB, integration, caching etc. A backing service*is* something that is external to the application but depends on it. (Oracle, MySQL, NoSQL, RabbitMQ, SMTP, Cache etc.). The app may depend on services provided and managed by third parties (Google API’s, New Relic etc.). |
| **Detailed Description**  For a cloud native application, both local and third party services are one and same. Both are attached resources and should be accessed with URL’s or configurations stored in Config Server.  It is normal for JEE application to read from and write to storage directly. But for cloud native applications this is a violation, instead it should treat storage as a backing service so that resources can be attached, detached and reattached as and when required.  **Key takeaways:**   1. Declare the application dependency for backing services with YML files managed as external configurations 2. The binding of the backing services should be done by the platform 3. Since application access these as services, implement the circuit breaker to make sure the errors / exceptions are handled gracefully in case of service unavailability   With binding done external to the application, now it should be easy to attach or detach the backing services easily without having to redeploying or restarting the application |
| **PCF Context**   1. CF provides managed services which are by default available for all to utilize (Service Registry, Config Server, SSO etc.) 2. CF provides service broker to author external services but can be made available as service instances managed by PCF 3. All services are available as attached resources and can be accessed using URL, locators, credentials with configuration parameters 4. CF provides restage command to apply the changes across all running instances |

# Logs

|  |
| --- |
| **Overview**  Every microservice application need not manage own logs. In cloud native applications, logs are treated as event streams. Instead of writing logs in application level files, the logs should be routed to a centralized logging system. The routing of logs should be handled by the platform. |
| **Detailed Description**  It’s usual for JEE applications to configure target location of the log files and control other parameters like log rotation, rollover policies etc. related to logs. But 12 factor applications never worry about routing or storage of its log output stream. Instead, each running process writes its event stream, unbuffered, to standard output stream. In java parlance we call it as STDOUT or console.  During local development, the developer views this stream in their system to observe and analyze the application behavior, flow and data details. In other environments (Dev, Test, Prod etc.) each application stream would be captured by the platform. The platform collates the entire stream from different applications and routes it to the target system / application for viewing and archival purpose. These destinations systems or application is not visible to or configurable by the micro service app but are completely managed by the platform. Basically log aggregation, processing and storage of logs are nonfunctional requirements for the platform and not the applications.  When the application is decoupled with log processing, storage etc.   1. The application code becomes simple 2. Easy to swap the processing in different way without modifying the code 3. Increase the scalability of application by running more than 1 instance of the application and the platform handling the routing of log and application contends with just logging the details   **Key takeaways:**   1. Applications have to just log the details and not worry about log aggregation, routing or processing 2. Finalize the standard log format and enforce its usage across all applications 3. Always use logging abstraction (SL4J, Logback) and do not use System.out.println 4. Use AOP for logging method start, end and data inputs for the methods |
| **PCF Context**   1. CF provides Loggregator component which aggregates and streams logs, metrics from user apps and system components 2. The combined stream of logs of all apps and metrics can be accessed via Firehose, events can be monitored with nozzle component and data can be streamed to external log / metrics management apps (Sumo Logic) |

# Dev/Prod Parity

|  |
| --- |
| **Overview**  Microservice applications should be developed and deployed in environments with matching configurations. Usually development, Test and Production environments have different scalability and reliability related profiles. But cloud platforms keep multiple application environments consistent and eliminate the pain of debugging environment discrepancies. |
| **Detailed Description**  When we have different profiles, configurations, settings etc. between different environments then there are possibilities that the application working in one environment may not work in some other environment. This affects the confidence of the teams and start putting more effort to ensure the applications work on other environments.  Cloud native applications apply rigor and discipline to environment parity so that the team is confident that the application will work everywhere. **The twelve-factor application does not use different backing services between development and production.** Sometimes adapters are used to abstract away the differences in backing services. But once simple incompatibilities crops up within environments, code that worked and passed tests in development or staging would start to fail in production.  **Key takeaways:**   1. Ensure that there is less / no gap between different environments 2. All environments should use the same type and version of the backing services 3. Use adapters only when applications are ported to new backing services. Once the porting is complete do away with adapters |
| **PCF Context**   1. CF provides “Spaces” to configure different environment with more similar configurations 2. It also provides plugins and integration with CI / CD tools thereby enabling development team to be actively involved in deployment activities |

# Disposability - Rapid Start and Graceful Shutdown

|  |
| --- |
| **Overview**  Microservice applications should have quick start and graceful shutdown features. Cloud processes are disposable i.e. it can be destroyed and created at any time. Designing the applications for rapid start and graceful shutdown helps to ensure good uptime, enables auto scaling easily and improve robustness of deployments |
| **Detailed Description**  Earlier the application containers would take more time to startup. This could be due both heavy components within the container or the applications taking time to startup due to processing done during the startup of the application.  In cloud scenario, even though application auto scaling enabled based on request load etc., if the container startup or application startup / shutdown are going take more time, then the application users are going to get affected since many requests may not be processed since the instance takes more time to startup. So if we need to rapidly bring up more instances to handle the load, any delay in startup can hinder the application ability to startup quickly.  Similarly if the application does not shut down gracefully it might result in take more time to recover from failure and also affects the instance startup. Sometimes this might also result in corruption of data, requests not being processed successfully and so on.  **Key takeaways:**   1. Always ensure not to perform any long running processing during the application startup 2. In case anything needs to done during startup of the application, see if they can be made as backing services and improve the performance 3. Always use light weight containers so that the instances can be started quickly |
| **PCF Context**   1. CF uses “Garden” a container technology which does startup or shutdown in micro seconds 2. CF proposes to support other container technologies like Docker, Rocket etc. |

# Admin / Batch Tasks

|  |
| --- |
| **Overview**  Microservice applications should delegate admin & batch jobs execution to platform layer. Sometimes there could be situations where we might to run timer jobs, one-off scripts or run interactive shells for performing some operations or executing processes. In those situations it is better to see if this can be dealt in some other manner or utilize the platform provided capabilities for performing the same |
| **Detailed Description**  Following are some of the situation which can be termed as Admin processes   1. Running scheduled jobs, for particular timings, repeated run every hour etc 2. Run a job only once for some processing   The problem with timers / schedulers configured within the application is when we have to scale the application, and then many instances might start processing at the same time. This might result in corrupting the data or adding additional logic to overcome this issue which might violate the 12 factor app principles  **Key Takeaways:**   1. Never launch jobs / processes from within the application 2. Try to expose those processes as RESTful endpoints so that it can be configured to invoke it in cloud native manner 3. Move the batch processing related code and make it as a separate microservice. Then time the service to be called at required times with a scheduler outside the application 4. Look at possibilities of utilizing the platform features |
| **PCF Context**   1. CF provides Tasks to perform one-off jobs. CF Tasks run for a finite amount of time and then stops. It runs in its own container which gets destroyed after running the task 2. Following are some Task related commands    1. cf run-task    2. cf tasks    3. cf terminate-task |

# Port Binding

|  |
| --- |
| **Overview**  Microservice application should delegate Post Binding activity to platform layer. It’s typical to see several applications running in the same container, applications separated by port numbers and using DNS to provide a friendly name to access. But when in cloud environment we do not need perform these and instead leave it to the cloud provider or the platform to handle it |
| **Detailed Description**  **In non-cloud environments, the applications are deployed to containers like Tomcat, JBoss, WebLogic and WebSphere application server. The containers takes the responsibility of assigning the ports for these applications when it starts up**  **But cloud native applications a completely self-contained. It exports HTTP as service by binding to specific port and starts to listen on that port to receive the requests. The platform with a routing component would handle the requests and maps it to that particular web processes. For example, development environment it might run on localhost:12000, for Test environment it might run on 10.10.10.10:15000 and in production with proper domain name convention application.company.com.**  **Applications developed with port binding externalized allows**   1. **The platform handle environment specific port binding with any code change** 2. **This helps services to acts as backing services for another application or services**   **Key Takeaways:**   1. **Externalize and export the port binding so that the platform handles assigning of port at runtime** 2. **Utilize Web as starter template which includes Tomcat as well as part of the overall application** 3. **Utilize named routes rather than random routes** |
| **PCF Context**   1. All services are available through URI and may not require any runtime injection of Versions or Package details. 2. The application code has dependencies to the run-times necessary to expose its services/ports, and CF mandates that the build-pack has the server libraries required to expose any services |

# Processes

|  |
| --- |
| **Overview**  Microservice applications should always be stateless. In case state information need to be persisted then external cache stores (like REDIS / PCC) can be used as backing service. Cloud native applications / processes follows stateless and share-nothing model. |
| **Detailed Description**  In non-cloud environment the web applications used store some data in session for different purposes. With clustered instances, the different servers provided different mechanisms to make these data accessible by the applications deployed across the clusters.  With cloud native applications, it is not that application should not use state at all. It is such that states cannot be maintained within the application. It should be external to the application provided by backing service  **Key takeaways:**   1. Always ensure the applications / services remain stateless 2. In case there is a need to maintain state, make it external provided by backing service like MySQL / Redis / PCC as per the scenario. |
| **PCF Context**   1. Any state that needs to be shared or persisted is done though the PCF provided managed service cache / database instances 2. Cloud Foundry normally kills current processes and starts new one. It never allows update to a specific process and hence the state is never shared |

# Concurrency

|  |
| --- |
| **Overview**  Microservices applications should support horizontal scaling by leveraging hosting platform capabilities. The microservices hosting platform handles diverse workload by assigning each type of work to a specific process type (Web request handled by Web process, long-running background tasks handled by worker process etc.) |
| **Detailed Description**  **In non-cloud environment we normally perform vertical scaling, adding more CPI’s, RAM and other resources. With cloud IaaS, it was made easy to scale out these sort of resources. But when it comes to containers, cluster of container farms were created to manage the load the applications deployed as part of these containers. But still it was difficult and complex to maintain the clusters.**  **Cloud native applications which follows the share-nothing, horizontal partitioning practices makes adding more concurrency in a simple and reliable manner**  **Key takeaways:**   1. Give at most importance to design considerations like stateless, share-nothing and disposability 2. Utilize the platform’s support for auto scaling, blue green deployments etc.   This then helps the application to take full advantage of horizontal scaling and running multiple concurrent instances of the application |
| **PCF Context**   1. CF encourages shared-nothing, horizontal partitioning of work loads 2. CF provides the support to perform scaling in following ways    1. cf scale    2. Auto Scaling Service with built in parameter    3. Write custom service to auto scale instances based custom parameters |

# Beyond 12 Factors – API First

|  |
| --- |
| **Overview**  When multiple teams develop services which need to interact with each other and having different release calendar, it becomes more complex and eventually would result in integration failures. API First approach helps to avoid integration failures and gives teams, the ability to work with each other using contracts without interfering with the development process |
| **Detailed Description**  API first refers to building API to be consumed by client applications and services. A user interface like web or mobile apps are consumers of an API.  Designing API first helps facilitate discussion with all stakeholders (internal team, business team, or possibly other teams within your organization who want to consume API). Collaboration with different teams allows to build user stories, mock API's and generate documentation that can be used to further socialize the intent and functionality of the service under development.  This is called as contract-first development pattern, where developers concentrate on building the edges or seams of the application first. With the integration points tested continuously, 2 teams can work on their own services and still maintain reasonable assurance that everything will work together properly. This allows rapid prototyping, support services ecosystem, and facilitates automated deployment, testing and continuous delivery pipelines  **Key takeaways:**   1. Design API’s first before starting the development 2. Document the API’s so that the API usage is clearly stated and easy to understand by consumers of the API 3. Utilize security tools like Fortify etc. to uncover any security loopholes within the application 4. Ensure security testing I carried out as part of any major release 5. Enable auditing to understand and identify the modification of data |
| **PCF Context**   1. Provides integration with different API Gateways products (APIGEE API Gateway, CA API Gateway etc.) |

# Beyond 12 Factors – Telemetry (Monitoring)

|  |
| --- |
| **Overview**  Building applications on desktops makes it easy to inspect the application, execute debugger and perform tasks that give visibility deep within the app and its behavior. But in case of cloud application there is no direct access and this presents an unfamiliar pattern for real-time application monitoring and telemetry |
| **Detailed Description**  Auditing and monitoring cloud applications are often overlooked, but it is very much important to plan and do it properly. While monitoring applications in cloud , the data that is sought for, falls under below categories  Application performance monitoring (APM):  Helps monitoring application performance. (Ex: HTTP Request performance)  Domain-specific telemetry  Helps monitoring business specific data ( Ex: Number of claims submitted, processed)  Health and system logs  Helps monitoring application infrastructure (Ex: Server status, JVM Memory Usage)  **Key takeaways:**   1. Plan the monitoring strategy and the parameters that needs to be monitored 2. Ensure auditing and monitoring is enabled for production environment 3. Configure notifications in case of any failure or exception |
| **PCF Context**   1. Provides Pivotal Cloud Foundry Metrics to monitor Container, Network, Application specific events, Search Logs etc. 2. Provides options to integrate with other logging, monitoring products (New Relic, Dynatrace, Sumologic etc.) |

# Beyond 12 Factors – Security First Design

|  |
| --- |
| **Overview**  Microservice applications should consider security implementation at multiple levels (app / data / network). Most of the times application security is not taken that seriously and it’s always an afterthought. Security should be a vital part of any application cloud environment or platforms |
| **Detailed Description**  Cloud native applications runs in different data centers, within multiple containers and is accessed by many users. Hence security should never be an afterthought.  Cloud native applications usually secure all the endpoints with role based access control (RBAC). Every request to the application resource should be checked and validated to understand who is accessing the resource and does he has the appropriate privilege and role to access the resource.  Today many options exist for handling the security, like OAuth2, OpenId Connect and SSO servers. Utilize the SAML / JWT mechanisms provided by the security provider to secure the application accordingly.  **Key takeaways:**   1. Secure each and every end points with specific authorization resource 2. Follow secure coding best practices 3. Utilize security tools like Fortify etc. to uncover any security loopholes within the application 4. Ensure security testing I carried out as part of any major release 5. Enable auditing to understand and identify the modification of data |
| **PCF Context**   1. CF provides Single Sign On as Managed Services for the applications to use for Authentication and Authorization purpose 2. Create specific service with granular access level for resources for each of the environments |

# Appendix