HCP AND CF@HCP FROM AN APPLICATION PROVIDER'S POINT OF VIEW

Authors: Christian Fecht, Petio Petev
P&I Technology, HCP

Status: Version 1.0
Nov 26, 2015

ABSTRACT

In this paper we describe the differences between today's HCP and the emerging CF@HCP from an application provider's point of view. With CF@HCP we denote the renovated implementation of HCP leveraging the open source PaaS implementation Cloud Foundry. We look at the three most important activities of an application provider: (1) develop an application; (2) operate an application; (3) sell and provision an application to application consumers, that is, the customers of the application provider. Moreover, we try to provide answers to the three most pressing questions and concerns of application providers regarding CF@HCP: How many new skills does CF@HCP require from me? How can I move my existing applications, possibly with their customers, from HCP to CF@HCP? What is the best way to develop a new application on HCP to ease the future transition to CF@HCP once CF@HCP will be available? It is especially this last question which we will try to answer in this paper.

TABLE OF CONTENTS

INTRODUCTION

With SAP HANA CLOUD PLATFORM (HCP) we have a PaaS product on the market, actually since 2012, which is used by application providers (SAP, partners, customers) to develop, run, operate, and ultimately provide applications to their customers (application consumers). In 2014, we decided to radically change the implementation of HCP by using the open source PaaS implementation Cloud Foundry and leveraging the Cloud Foundry community. Internally, we use the term CF@HCP to denote the renovated version of HCP based on Cloud Foundry. The fact that CF@HCP will be based on Cloud Foundry can and should not be hidden, that is, application providers will see differences between HCP and CF@HCP. Needless to say, that application providers who have invested in HCP and who have been using HCP since quite some time have valid concerns and questions, which basically boil down to the simple question: How new and how different will CF@HCP be for me? Going into more details, application providers have typically the following three questions: Which new skills does CF@HCP require? How can I transition my existing HCP applications, possibly with their productive customers, to CF@HCP? How can and should I develop new applications on HCP to ease the future transition to CF@HCP once CF@HCP will be available for productive usage?

In this paper we'll try to answer these questions by having a systematic look at the main activities (develop, operate, provide) of an application provider and how CF@HCP will change them. We will also provide a recommendation how to develop applications on today's HCP to ease the future transition to CF@HCP.

THE APPLICATION PROVIDER

The application provider is the single most important persona for HCP. He is actually the one who uses HCP as a full-fledged platform. He develops applications on HCP. He runs applications on HCP. He sets applications productive on HCP. He operates applications on HCP. And, most importantly, he provides and sells applications to his customers and application consumers. Actually, providing and selling applications to customers is the major goal of any application provider and all his other activities, like developing or operating applications, are just a means to achieve this goal. We will divide the activities of the application provider into three major categories

- *Develop* an application
- *Operate* an application
- *Provision* an application

and will analyze how CF@HCP will impact and possibly change these activities. But, before looking at CF@HCP, let's first describe in more detail the three activities *develop*, *operate*, and *provision*.

Develop Applications

Developing application is typically all about code – and a touch of configuration – and comprises typical developer activities like coding, deploying, testing, debugging, profiling and so on. The actual code a developer writes and which makes up the application depends on several factors:

- runtime of your language, e.g., Java SE 8
- application container, e.g., Tomcat 8
- available libraries, e.g., Spring
- available remote services
- contract between runtime and services

A standard web application built using Java SE 8, Tomcat 8, Spring 4, and using JDBC / JPA to connect to a database has little or no dependencies to how the underlying platform manages application server processes and should run actually in any environement providing Java SE 8, Tomcat 8, Spring 4, and JDBC / JPA.

Operate Applications

Once your application is productively used by an application consumer, you have to operate it appropriately. Once in a while you have to upgrade your application to a newer version, if possible, without any downtime for your customers. You need to have monitoring in place to see the health of your application. You want to get alerts if your application is in trouble. And because your application is actually a cloud service, you need to know the service quality (availability, SLAs) of your application. Often, applications are monitored and operated using external tools with their own processes, and the platform must enable you to integrate the operation of your application into those external tools and processes.

Though operating has some connection to coding (logging, providing custom monitoring), it is mostly not related to coding and developing. It is determined by the capabilities of the platform, for instance, supporting application updates without downtime, and the tools offered by the platform: log viewers, monitoring dashboards, SLA dashboards, configuration of alerts and mail

notifications, and the availability of analytics on operation relevant data.

Provision Applications

In most cases applications are written to be used by third parties. For symmetry we call them application consumers. In order to expose an application for consumption, application providers have to keep the notion of "subscription". Based on subscription metadata, consumers can be given a dedicated instance of an application or use tenant aware instance(s) of the application in the bounds of specific quota defined by the subscription. To enable application consumers to buy, i.e., subscribe, to an application, it has to be modeled in a marketplace with description, price, and business model. A dedicated infrastructure, the commercial infrastructure services in the case of HCP, implements the technical steps needed to "provision" the application by creating the needed configuration, metadata and possibly new instances of the application. The notion of subscription is pervasive:

- It determines whether an application consumer has the permission to use certain resources of the application
- It is used as anchor for application consumer specific configuration, e.g., identity provider, role mapping
- It is used to maintain and transparently pass the application consumer's identity (tenant) and make it available as discriminator in multi-tenant scenarios

The ability of application providers to provision their applications to application consumers can be summarized by the following capabilities: defining application metadata for publication on a marketplace, defining technical steps needed for provisioning, maintaining and accessing subscription level metadata at runtime.

FROM HCP TO CF@HCP

With the adoption of Cloud Foundry in HCP we offer a modern polyglot PaaS to our application providers enabling them to use modern programming models for writing applications. The Cloud Foundy programming model, that is, the Cloud Foundry way of building applications, will eventually become the default model in HCP. From an application programming point of view, the main benefits of adopting Cloud Foundry are:

- open HCP to new programming languages and technology stacks for application development and enable application providers to choose the technology that best fits their skills and needs.
- offer an open standard for application development to internal stakeholders, partners, and customers
- embrace and leverage the Cloud Foundry developer ecosystem to offer well-known and widely-adopted development methodologies and tools instead of relying only on home-grown development flows.

HCP has been developed in an "evolving" manner since 2010. Cloud Foundry concepts were established around that time and have evolved in parallel. While there are overlaps and similar concepts in HCP and CF, they can be perceived as completely different and complementing each other. To distinguish between today's HCP and Cloud Foundry (and the future CF@HCP) we will use the adjective "classic" to refer to today's HCP and its programming model. In this chapter we will dive into some differences between HCP classic and CF@HCP from the point of view of an application provider.

Developing Applications

As discussed earlier in the document, application development is mostly around code. Standards like Java, Java EE, and de-facto standard open source frameworks and libraries provide already a good deal of portability and mostly abstract from the underlying infrastructure. If we consider only application code that uses such standards and 3rd party libraries (standard or not), the difference between applications running in the classic model and those running in the Cloud Foundry model is virtually non-existing. Considerable differences though exist when we put aside the mere business logic code of an application. Design time, service consumption model, and multi-tenancy capabilities are different between classic HCP and Cloud Foundry.

The following tables show the most significant differences between HCP and CF@HCP as of today. For the sake of simplicity we consider Java only.

HCP Classic	CF@HCP
Service clients / APIs are part of the runtime, mostly "Java only"	Service clients / APIs need to bundled with the application or come with particular build packs (see for example the Java buildpack)
Connecting to services is done via abstractions (destinations, data sources …) registered in JNDI	Connecting to services is done via low-level connect information in OS environment variables; there might be "helper libraries"

	coming with the buildpack or developed separately to facilitate the parsing and interpretation of the OS environment variables
Service connection authentication is transparently done	Service connection authentication is done manually via credentials stored in the environment variables
Applications cannot consume services provided by CF@HCP (unless there is network connectivity and the services of CF@HCP provide a language- and PaaS-agnostic API or expose native APIs for which a driver exists)	Applications cannot consume services provided by HCP classic (unless there is network connectivity and the services of classical HCP provide a language- and PaaS-agnostic API or expose native APIs for which a driver exists)
Support for Java EE oriented Eclipse WTP development flow; some configuration tasks enabled in Eclipse; proprietary command line tool for Java only	STS (Spring Tool Suite) supports Cloud Foundry. It is tailored towards Spring applications. Standard command line tool (Cloud Foundry CLI) which can be used for any type of application
Profiling and debugging capabilities for cloud-deployed applications	Local development recommended. Debugging and profiling is only possible locally
There is notion of tenant ID that is automatically inserted in the context of execution	No built-in notion of tenant; static configuration or other information like domain of the URL should be mapped to tenant specific metadata

The above differences can be address in two ways: (1) By providing a backward compatibility layer in CF@HCP for HCP classic or (2) by mimicking Cloud Foundry in HCP classic by providing a forward compatibility layer so that application providers can use the Cloud Foundry way of application development in HCP classic or have at least a less painful transition to CF@HCP later on.

A backward compatibility layer would mean that we create a proprietary Cloud Foundry "buildpack" for Java, where the current HCP classic service APIs exist; this is not a trivial task as things like "destination", "subscription" and "tenant" do not exist in the Cloud Foundry programming model today. Enabling them in Cloud Foundry would be quite artificial. Moreover, this approach would actually not help application providers to change to the Cloud Foundry model, it will merely enable deployment of existing applications unchanged in CF@HCP. It can be considered a short-term win that adds to technical debt in both the application and the platform.

The alternative is to expose Cloud Foundry constructs and entities in HCP classic via a forward compatibility layer allowing for early adoption of concepts that would work in Cloud Foundry:

- service bindings for existing HCP classic services are additionally be modelled as environment variables, in exactly the same way as in Cloud Foundry; from consumption point of view, this can also present CF@HCP-only backing services so they can be consumed from classical HCP applications
- applications should not rely on JNDI bindings for services
- destinations are also added to the environment variables in HCP classic

Although some gaps can be addressed by a backward compatibility layer or with Cloud Foundry constructs in HCP classic, other features in HCP will be missing in the Cloud Foundry environment. They are a mix of runtime and design time capabilities: multitenant configuration, debugging and profiling. Alternatives for those can be implemented for Cloud Foundry, with considerable additional investment.

Some features (like dynamic configuration and destinations, debugging and profiling) could make sense to be contributed to Cloud Foundry ecosystem overall.

In many cases it is better to use Cloud Foundry native tools, hence developers should learn as early as possible how development is done in a Cloud Foundry environment. For example – Eclipse development with Cloud Foundry is better done with STS and not with a plain WTP server adapter. Additional Eclipse Tools for HCP classic model would not give much benefit when developing in the Cloud Foundry environment. Or even better – learn how to depend only on the Cloud Foundry command line tool CLI for pushing applications to the cloud.

Operating Applications

Operations capabilities are another core domain of the platform. The actual application programming model is only loosely coupled with those operations capabilities and application code itself has little to do with them. Applications written in HCP classic can leverage all HCP classic operations capabilities while applications written for Cloud Foundry cannot. For many operations tasks there will be alternative tools for CF@HCP and a long term goal is to reuse some of those newer implementations of operations tools also for HCP classic if we prove they are better suited for the cloud compared to the current solutions. Like with development of applications, new tools and terminology have to be learned by application providers to be able to operate productive applications on CF@HCP. The following table illustrates the differences between tooling and capabilities around operation.

HCP Classic	CF@HCP
Monitoring and Alerting part of Cloud Cockpit	3rd party *(under construction)* Grafana & Influx DB
Log viewer part of Cloud Cockpit	3rd party *(under construction)* Elastic Search, Kibana, LogStash (called ELK stack for brevity)
ZDM update for Java apps in neo command line and in LM lifecycle service (rolling update and blue-green-like algorithms)	ZDM update for Java apps has to be implemented manually with blue-green deployment, however, there's tools like a CLI plugin that automate this[1]
Application availability dashboard: internal "uptime" application and external tools integration possible	Same as HCP classic
Configuration tools in neo command line and Cloud Cockpit	Configuration tools in CF CLI (command line interface)
Configuration can be updated without application restart	Configuration is static and application restart is needed which is usually mitigated by a blue-green update
Multi-tenancy support enables subscribing new application consumers without restarting applications	New application must be created and started when new application consumer wants to use an application
e-mail notifications can be sent with custom availability checks defined	*(under construction)* Similar things should be possible with 3rd party tools
Audit logs written and indexed by Splunk; not available to customers	*(under construction)* Audit logs initially in ELK stack, ideas for storing additionally to Cassandra and wrapping with reliable messaging
Applications lifecycle and application "processes" inside cockpit and via neo command line	*(under construction)* Similar support in cockpit; intrinsic support with CF CLI
HCP classic account members (users) and their roles managed in cockpit	Intrinsic support with CF CLI; idea to use the same identity provider as HCP; roles and permissions will be managed in a different way
Multi-tenancy support in application URL (tenant derived using host header)	Multi-tenancy can be achieved by manually defining routes to new applications for new tenants
Different versions of applications can be only managed by having new application name	Versioning is not part of core Cloud Foundry, so same approach should be used
Separation of dev, test, staging and productive deployments done via creating separate HCP classic accounts	Same approach as HCP classic, using CF spaces instead of HCP accounts
Updating to a new version of the Java runtime can be done with restart of the application processes	Updating to a newer version of the Java buildpack (similar to Java runtime) needs application redeployment but could be done by again applying blue-green update.

Management of application configuration is a special case and open question with CF@HCP. HCP classic has certain configuration in different levels – account, subscription and application. To

[1] https://github.com/bluemixgaragelondon/cf-blue-green-deploy

have the same capabilities in CF@HCP we have yet to choose a configuration server that can be used by applications. Cloud Foundry core components do not include such server and the only possible way of configuration is to push user defined environment variables when applications are deployed or to store that data in an application specific way in a backing service.

As a summary – operating applications is similar, but yet done with a different toolset when we compare HCP classic and CF@HCP. In addition, as operations tools for CF@HCP are in development, they can differ in future if there is a proof that the implementations under construction are not good enough or we invest in changing their look and feel.

Provisioning Applications

Once an application is ready for shipment to customers it is required to "expose" it in a way that it is consumable with the help of self service. HCP classic is integrated into SAP's order-to-cash processes and the corresponding systems. The most important of these systems is the ICP, the CRM system of the company SAP SE. All orders by SAP customers end up and are centrally managed in ICP. Application providers design their own business model and pricing policy which are transferred to ICP. Whenever a "tenant provisioning" request is generated for a solution built on HCP, ICP contacts the so called "Commercialization Service" in HCP which in turn is responsible to create a subscription from the customer's HCP account to the respective application. In this case "provisioning" an application is done merely by creating such a subscription and registering a URL that the customer can use. Another way of provisioning applications to customer is to actually deploy the binaries of an application in the customer's account which will require that the customer additionally has enough virtual machines quota to start the application. Both flavors of provisioning applications in HCP classic are automated by "Commercialization Service". The same automation mechanisms can be used by partners to sell applications built on HCP classic.

In comparison – today's CF@HCP does not have such integration with SAP's order-to-cash process and with ICP and respectively – automation of provisioning. Multi-tenancy in CF@HCP is implemented by starting new processes hosting the application and setting up respective authentication and authorization parameters. There are options to enable end to end provisioning by either extending the "Commercialization Service" to support Cloud Foundry applications, or to integrate and interoperate with Hybris YaaS platform approach for application/service provisioning.

GUIDELINE FOR DEVELOPING APPS ON HCP CLASSIC FOR LATER TRANSITION TO CF@HCP

Cloud Foundry introduces an modern application programming model that is aligned with today's needs for massive scalability in the cloud. This is not only done by offering features that application developers can use – there are certain limitations and constraints that, when respected, promise a scalable and resilient application which is easy to maintain. Cloud Foundry is tailored towards the state-of-the-art design principles of "12 factor applications" (http://12factor.net/) and microservices (http://bit.ly/1dI7ZJQ) for building cloud-ready applications built on top of self-contained services In this context, it is recommended to refactor existing HCP applications and design new applications written on top of HCP in a way that they fit in the constraints introduced by those principles. An additional and important aspect to consider when writing applications on HCP classic is to make them portable as much as possible. Below are some concrete suggestions for making applications more "Cloud Foundry-friendly", generally more resilient and scalable, and finally more portable:

- Applications should be **stateless**, all state should be externalized to some persistent or transient storage or, in case we have a disposable state needed by the client of the application (e.g. browser), to store the state on the client. Application processes have to be disposable at any given moment, keeping state in the process memory introduces risks for data loss. Restarting of processes in Cloud Foundry is much more common and cheaper compared to HCP classic where every application process runs on a dedicated own virtual machine.
- **Local caching** should be used with caution. Today we do not offer remote caching service in HCP classic, but such will be available in Cloud Foundry, for example by using Redis as a backing service. Local process caches are valuable for single node applications, but in case we scale application to more processes or VM's it becomes an issue in case we want to synchronize caches or remove stale or obsolete data from the

cache. Additional complexity is introduced to handle "split brain" syndrome in applications. This issue becomes more apparent with increasing the number of processes.

- Always run **more than one instance** of the application – this will prevent "centralized" logic to leak in the architecture; such centralized logic later is a promise for scalability failure. It is possible to run several instances of the same application in HCP classic – use that capability from the beginning. In case centralized logic is still needed (e.g. scheduling a one-time job), it is encouraged to use already existing libraries that externalize state in persistent store.
- Avoid designs with **point to point communication**; processes in Cloud Foundry cannot communicate with each other directly. This is also not encouraged in HCP classic as leads to problems with managing clusters and issues with scalability. In the same context – avoid having primary/secondary instances of the application – building consensus and maintaining a running primary instance is complex and discouraged, use backing services instead (e.g. messaging, persistent store).
- Build your app using **"API first" approach**. Model functionality with REST APIs and oData APIs; in distributed architecture the remote API is the most important asset of applications and services; never rely on GUI for abstracting functionality – it cannot be used programmatically and should be easily exchangeable in terms of technology and device form-factor.
- When building an application in distributed/microservice manner, always **use application REST API or API based on messaging**. Do not communicate via shared resources which would break semantic encapsulation (e.g. same database or database schema, same data model). This will prevent coupling between microservices and will increase cohesion (use of a microservice in the way it is supposed to be used).
- Do not **overdesign** the application with regards to separating to **microservices**. First of all – try to understand the business domain model and the lifecycle of entities inside the model. Based on that – create "bounded areas" (a.k.a. Bounded Contexts: http://bit.ly/1o7AK8B) which would characterize separated microservices.
- Model application features as resources and build **resource scopes** for particular scenarios (as defined by OAuth 2.0). This helps tremendously in several directions – flexible and tangible authorization control, REST API mapping and definition of business roles. OAuth authorization will enable cross consumption of APIs between HCP classic and CF@HCP, making migration of applications easier. Supporting OAuth will make applications and services consumable from external clients.
- Maintain an explicit list of **dependencies** while evolving your application or service, try to wrap dependencies in a "resilience" library like Netflix Hystrix (https://github.com/Netflix/Hystrix). This will help when dependencies have issues and will make your application / service fault tolerant. Adding a wrapper around dependencies will also allow easier porting when migrating from HCP classic to CF@HCP and in general, when changing endpoints of services, or the way they are configured.
- Strive to have **admin/configuration** API endpoints **different** from business logic endpoints: administration of the service itself and business logic has to be strictly separated. System level administration that is not oriented towards users of the applications (i.e. instead - oriented towards application operators) ideally should be hosted on a different process, which is only launched when administrative tasks are required.
- Do not rely on **local file system**, your application / service instances should be disposable. In the same context – do not couple with machine **IP addresses**, it is absolutely sure they will change.
- Portions of code where you read **configuration** or bind to platform service endpoints should be isolated in order to be not "locked in" too much in specific way of configuration and service consumption. All external endpoints (e.g. to 3rd party services) should be maintained in isolated "helper" classes. Again – Hystrix can be used for abstraction. Such endpoints (URLs) are considered configuration in HCP
- Try to consume external resources always **asynchronously**, either by using

a library like Netflix RxJava (https://github.com/ReactiveX/RxJava), or if the dependent backend supports it – via asynchronous messaging. This will allow massive scalability and will relieve the process and the operating system from managing many threads.

Following these guidelines will, to a big extent, make applications both "cloud-ready" and easier to adapt with regards to differences between platform flavours. If you use services of HCP classic, you should centralize and isolate their invocation to ease future migration to CF@HCP.

CONCLUSION

Let's summarize how the three main activities (develop, operate, provision) of an application provider will change when transitioning from HCP classic to CF@HCP. Operating applications in CF@HCP will be completely different because CF@HCP offers a completely different and more modern operation toolset to application providers. CF@HCP has so far little intrinsic support for helping application providers to provision their applications to their customers. In contrast, HCP classic provides the concept of subscription, with built-in multi-tenancy support, to easily establish and manage the relationship between application providers and their application consumers. Developing applications is the most complex of the three activities of an application provider. On the hand, a Java application running on a container like Tomcat has almost no dependencies to how application processes are managed by HCP classic and CF@HCP. Sure, there are differences between HCP classic and CF@HCP in how application get initial access to services. But this could be easily solved by adopting the Cloud Foundry contracts (OS environment variables, service broker architecture) in HCP classic. Another difference is the development toolset. HCP classic supports live debugging and profiling of applications which, to our knowledge, is not supported by CF@HCP. In addition, Cloud Foundry better supports the Spring-based approach for building Java applications. The biggest difference, however, is that Cloud Foundry and hence CF@HCP has been designed to support modern state-of-the-art design principles for building scalable cloud-ready applications and services. Though HCP classic does not fully support these principles, it provides sufficient support that application providers can and should already follow these principles on HCP classic to fully leverage them when later on transitioning their applications to CF@HCP.