**API FACTORY**

HOW TO

Centralize Configuration with ConfigServer and SpringBoot

**This page displays the U.S. Export Notice. Do NOT delete.**

**Table of Contents**

1 Initial considerations 3

2 How to… 3

3 Testing and Running locally. 6

3.1 Profile-specific properties 6

3.1.1 Standalone: 6

3.1.2 spring-boot-maven-plugin + maven-surefire-plugin: 6

4 Running with ConfigServer 6

5 Reading properties from ConfigServer 8

5.1 @Value Annotation 9

5.2 Environment 9

5.3 @ConfigurationProperties 9

6 Logging 10

# Initial considerations

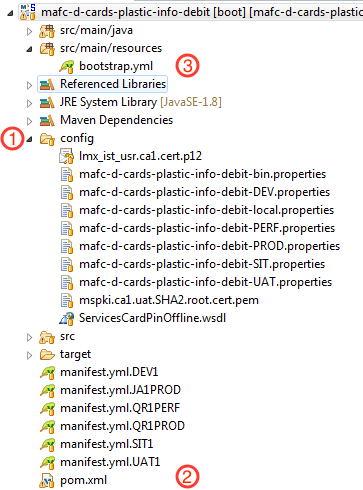
This tutorial will show how to load configuration files from different directories with Spring Boot and ConfigServer, how to test and run with Maven locally and the standards followed by the API Factory.

By default, [**Spring Boot**](https://docs.spring.io/spring-boot/docs/current/reference/html/) look for your externalized configuration file (application.proroperties) in four predetermined locations:

1. Classpath root, (src/main/resources or $project\_jar/classes after packaging the project)
2. Package /config in classpath,
3. Current directory
4. Subdirectory /config of the current directory.
5. ConfigServer serves configurations stored as either Java Properties files or YAML files. In Citibanamex, it reads files from a SVN repository (a *configuration source*). Given the URI of a configuration source, the server will clone the repository and make its configurations available to client applications in JSON as a series of propertySources. **Properties from a ConfigServer will override those defined locally (e.g. via a bootstrap.yml in the classpath).**

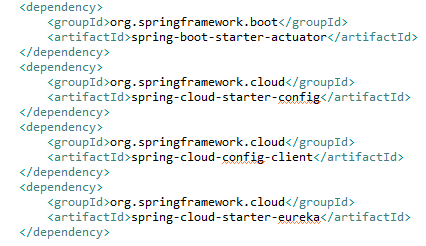
# How to…

In the API Factory we are following the number 4 to run and test locally and number 5 once pushing the app to PCF, so the below structure will be used as an example in further sections of this document:

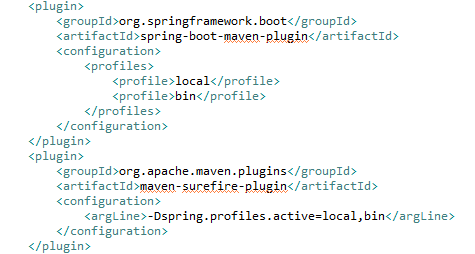


1. All configuration files and certs should be removed from src/main/resources folder and move them inside of the /config subdirectory.

2. Besides the initial configuration provided by the API Factory, the pom.xml must have the following configuration:



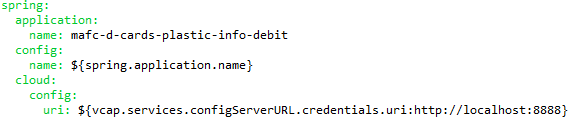
…



*Profiles depends on the project and you can have as many as needed.*

Dependencies will allow you to bind Microservices to the platform services during the deployment into PCF. The additional configuration in the spring-boot-maven-plugin will allow you to run with different profiles, loading properties according to the environment. This is explained in detail later.

3. The content of the bootstrap.yml file is shown in the image below, where the most important value is the one for spring.application.name:



# Testing and Running locally.

The first way to tell Spring Boot to load our configuration files is by using command arguments. Spring Boot provides the argument spring.application.name to set configuration file.

## Profile-specific properties

In addition to application.properties files, profile-specific properties can also be defined using the naming convention application-{profile}.properties

Profile-specific properties are loaded from the same locations as standard application.properties, with profile-specific files always overriding the non-specific ones irrespective of whether the profile-specific files are inside or outside your packaged jar.

If several profiles are specified, a last wins strategy applies. Based on the example, once you run the application locally, it will try to find the configuration from the configServer, but due to its absence, it will find then:

./config/mafc-d-cards-plastic-info-debit.properties

./config/mafc-d-cards-plastic-info-debit-bin.properties

### Standalone:

java -jar -Dspring.profiles.active=local,bin mafc-d-cards-plastic-info-debit-0.0.1-SNAPSHOT.jar

### spring-boot-maven-plugin + maven-surefire-plugin:

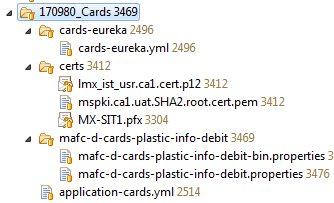
mvn test

mvn spring-boot:run

# Running with ConfigServer

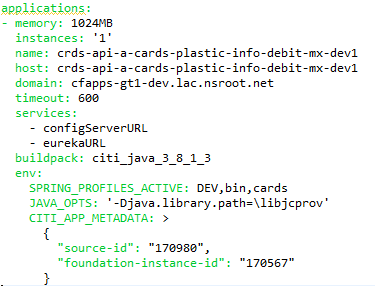
A configuration source contains one or more configuration files used by one or more applications. Each file applies to an *application* and can optionally apply to a specific *profile* and/or *label* (label approach hasn’t been applied in the API Factory yet).

The following is the structure of a SVN repository which is used as a configuration source:



In the image above, the configuration source defines configurations for the based example. The Server will serve different properties according to the application name mafc-d-cards-plastic-info-debit that will load the properties depending on the values of {profile} in the request path. If the {profile} is neither DEV, SIT, UAT, PERF nor PROD, the server will return the properties in mafc-d-cards-plastic-info-debit.properties, or if the {profile} is production, the server will return the properties in both mafc-d-cards-plastic-info-debit-PROD.properties and mafc-d-cards-plastic-info-debit.properties.

The following image shows the basic information to bind the Microservice to platform services in PCF:



The use of SPRING\_PROFILES\_ACTIVE environment variable will set the profiles as active, the application will request configurations by the path format:

/{application}/{application}-{profiles}.{yml|properties}

Therefore, during deployment you will see the following propertySources:

/mafc-d-cards-plastic-info-debit/mafc-d-cards-plastic-info-debit-DEV.properties

/mafc-d-cards-plastic-info-debit/mafc-d-cards-plastic-info-debit-bin.properties

/application-cards.yml

# Reading properties from ConfigServer

Spring provides different ways to read values from properties files:

* @Value annotation
* Environment
* @ConfigurationProperties

According to Citibanamex deployment architecture view, which separates the environment, there is no difference on using each of them, just some pros and cons by using each one.

## @Value Annotation

The first and simple way to read a value anywhere by using @Value. In order to do so, PropertySourcesPlaceholderConfigurer must exist in the application contexts that placeholders resolution is required.

The disadvantage of this approach is that every class which uses @Value should be updated if a property changed.

## Environment

The Environment approach is a combination of profiles and properties. That is the main difference in usage of Environment object to inject by @Value.

“*The role of the Environment object with relation to profiles is in determining which profiles (if any) are currently active, and which profiles (if any) should be active by default.*” – Spring’s docs

That means, with the **Environment** object, we can access the information related to profiles, but can not if using **@Value**. So, if we don’t need profile’s information, we should use **@Value** annotation.

“In most cases, however, application-level beans should not need to interact with the ***Environment*** directly but instead may have to have ${…} property values replaced by a property placeholder configurer such as ***PropertySourcesPlaceholderConfigurer***“ – [Spring’s docs](http://docs.spring.io/spring/docs/current/javadoc-api/org/springframework/core/env/Environment.html).

However, using **Environment** to get properties value is just like using **@Value** – increasing Spring coupling if we don’t have a separate class for properties.

To work with Environment, we use @Autowired:

@Autowired

Environment environment;

Another option is to get if from Application Context:

Environment environment = (Environment) context.getEnvironment();

Then, to read the individual property in application, we have getProperty method:

environment.getProperty("app.connection.url");

## @ConfigurationProperties

Spring Boot provides this annotation to work with properties that allow complex beans and validate the application configuration. This approach helps us to separate configuration by properties classes, so property values can be bound to structured objects conveniently.

# Logging

To fulfill Citibanamex business requirement, the API Factory proposed the following best practices for logging microservices running on a distributed architecture.

In the microservices world, it is common to have multiple instances of the same component running at the same time. That’s why it is important to have an instance identifier on the log entry that shows where that entry came from.

Hence **first proposal** to use logging, is to expect a **unique identifier (UUID)** from the request, i.e. the customer should generate a unique identifier for each transaction that can be used later to correlate events and trace transactions easily. Once the development has started, be sure to find the **citibanamex-spring-boot-starter-logging** dependency in the pom.xml, this will initialize the logging in the Microservice and add the uuid automatically in each request.

Secondly the API Factory exhorts to the developers to understand the following reading about the use of logging.

<http://www.masterzen.fr/2013/01/13/the-10-commandments-of-logging/>

Here in Ccitibanamex, we defined to handle three levels of logging:

**DEBUG:** this will help to trace requests by meaningful messages, as per the reading said. So please don’t get confused when tracing messages inside a loop or specific block of code and avoid things like the below:

**if** (cardDetails.equalsIgnoreCase("credit")) {

***logger***.debug("I’m here ..");

}

**INFO:** Add this level to messages that refer to important process by Business side. For instance when it is going to apply a withdrawal or deposit, a payment, an update for a PIN, and so on.

**ERROR:** We should be able to know when and where occurs an error and the reason about it.