Spring Profiles

Otimizando configurações

Fernando Teixeira Alves de Araujo

Agenda

- Introdução
- application.yaml e application.properties
- Inicializando Aplicação com Profile
- Property PlaceHolder
- @Value e @ConfigurationProperties
- @Profile
- @ActiveProfile
- @SpringBootTest(properties={“app.name=api”})
- @DynamicPropertiesSource
- Cases
- EnvFile
- Q&A

Introdução

- Spring Profiles fornece uma forma de **segregar partes das configurações** da sua aplicação e torna disponível apenas em um **determinado ambiente**. Quaisquer das **anotação** @Component, @Configuration or @ConfigurationProperties podem ser marcada com @Profile para **limitar o profile** quando é carregado (docs.spring.io 2024)
- Profiles é uma **funcionalidade core** do spring — nos permitindo **mapear nossos beans** para **diferentes profiles** — por exemplo, **dev, test, and prod**. Então nós podemos a**tivar diferentes profiles** em **diferentes ambientes** para carregar na inicialização apenas os beans que precisamos. (baeldung.com 2024)
- O uso de Profiles foi primordial para a criação de **Starters Auto Configuráveis**, considerando que para adicionar um Starter é necessário apenas **adicionar a dependência** do starter no pom.xml e **configurar no application.yaml/application.properties**.

application.yaml e application.properties

- O application.yaml/application.yml/application.properties é o arquivo responsável por manter todas as **configurações** dos **starters** e **beans** utilizados por nossa aplicação;
- O Spring Boot está **preparado** para ler os formatos **YAML** (*.yaml e *yml) e **PROPERTIES** (*.properties);
- Para configurar diferentes profiles, é necessário colocar o **sufixo iniciado por hífen mais o nome do profile** no nome do arquivo, por exemplo:
  - application-prod.yaml - Isso significa que essas configurações pertencem ao profile **“prod”**
- Porém, se **não** for colocado um nome de profile, como application.yaml, isso significa que essas configurações vai representar o **profile default** que por sua vez vai ser a **base para qualquer profile** configurado

Inicializando Aplicação com Profile

Para **ativar** um ou múltiplos na inicialização de uma aplicação, podemos fazer das seguintes formas:

- Alteração da propriedade `spring.profiles.active` normalmente no a**rquivo de properties com profile default**:

```
spring.profiles.active=dev,hsqldb
```

- Alteração da propriedade nas propriedades passadas por **parâmetro na inicialização** do JAR:

```
java -jar -Dspring.profiles.active=dev,hsqldb application.jar
```

- Criação de uma **variável de ambiente** com o nome `SPRING_PROFILES_ACTIVE`

```
export SPRING_PROFILES_ACTIVE=dev,hsqldb
```

Property PlaceHolder

- Property PlaceHolder são **valores dinâmicos** que atribuir um valor em t**empo de execução** na sua aplicação.
- A **sintaxe** de um Property PlaceHolder é `${NAME:ALTERNATIVE_VALUE}`, onde name pode representar o nome de uma **Variável de Ambiente** do Sistema Operacional ou uma outra **propriedade já configurada** no mesmo profile ou em algum dos outros profiles que estão ativos e, se **não existir** a variável de ambiente ou a propriedade mencionada, o **valor alternativo é atribuído à propriedade**.

Property PlaceHolder

Exemplos:

- Usando variável de ambiente:

  `${JDBC_URL}`

- Usando variável de ambiente com valor alternativo:

  `${JDBC_URL:jdbc:h2:mem:testdb}`

- Usando outra propriedade:

  `${spring.datasource.url}`

- Usando outra propriedade com valor alternativo:

  `${spring.datasource.url:jdbc:h2:mem:testdb}`

Property PlaceHolder

- Também podemos criar Property PlaceHolder custumizados, como exemplo: leitura de valores de um Key Vault (ASW KMS ou GSM), de arquivos acessados via HTTP ou de arquivos locais.
- Exemplo de um Property PlaceHolder para abrir um arquivo local:

```
${file://${DATABASE_PASSWORD:C:/database/password.txt}}
```

@Value e @ConfigurationProperties

- Para obter valores e beans a partir de **configuração das propriedades**, podemos usar as anotações: `@Value` e `@ConfigurationProperties`, sendo que:
- @Value é usado para obter o valor de um **único property**
- E o `@ConfigurationProperties` serve para pegar valores de **um conjunto de properties** correlacionado.

@Value e @ConfigurationProperties

- Exemplo do uso da anotação `@Value`:

```
@Component
public class MainConfigProperties {

    @Value("${smtp.mail.hostName}")
    private String hostName;
    @Value("${smtp.mail.port}")
    private int port;
    @Value("${smtp.mail.from}")
    private String from;

    // standard getters and setters
}
```

@Value e @ConfigurationProperties

- Exemplo do uso da anotação `@ConfigurationProperties`:

```
@ConfigurationProperties(prefix = "smtp.mail")
public class MainConfigProperties {

    private String hostName;
    private int port;
    private String from;

    // standard getters and setters
}
```

@Profile

- A anotação `@Profile` é usado para **instanciar um objeto** no contexto do Spring a partir de uma configuração de forma **condicional**:
- Exemplo:

```
@Profile(“local”)
@Configuration
public class LocalDatabaseConfig {
                                    @Bean
    public String jdbcUrl() {
        return “jdbc:h2:mem:testdb”
                                    }
}
```

@Profile

- Também podemos fazer a **negação do condicional** feito na anotação `@Profile` incluindo o **ponto de exclamação** antes no profile.
- Isso faz com que a instância só seja colocado no contexto de spring se o profile for **diferente no profile negado**.
- Exemplo:

```
@Profile(“!local”)
@Configuration
public class DatabaseConfig {
    @Bean
    public String jdbcUrl() {
        return “jdbc:postgresql://postgresql.company.com/test”
    }
}
```

@ActiveProfile

- A anotação `@ActiveProfile` é utilizado em conjunto com a anotação `@SpringBootTest` e é utilizado para **forçar um profile** na execução de um **teste unitário**.

```
@SpringBootTest
@ActiveProfiles(value = "test")
public class TestActiveProfileUnitTest {

    @Value("${profile.property.value}")
    private String propertyString;

    @Test
    void whenTestIsActive_thenValueShouldBeKeptFromApplicationTestYaml() {
        Assertions.assertEquals("This the the application-test.yaml file", propertyString);
    }
}
```

@SpringBootTest(properties={“app.name=api”})

- Sabendo que a anotação `@SpringBootTest` é usado para rodar um **teste unitário subindo o contexto do Spring**, quando usamos o campo properties da anotação, **forçamos o valor de um propriedade** onde só tem visibilidade no teste unitário.
- Exemplo:

```
@SpringBootTest(properties = {"app.name=api"})
class ApplicationPropertiesTest {

    @Value("${app.name}")
    private String appName;

    @Test
    void testAppNameProperty() {
        assertThat(appName).isEqualTo("api");
    }
}
```

@DynamicPropertiesSource

- A anotação `@DynamicPropertiesSource` é aplicado a **testes unitário** e é usado quando temos que atribuir um valor de uma propriedade que é **dinâmico** e só temos os valores em **tempo de execução**.
- Um exemplo clássico é quando usamos Testcontainer e o container cria um **nome de host** ou **porta** de forma aleatória.
- No exemplo a seguir vemos esse exemplo de uso de um Testcontainer iniciando um PostgreSQL:

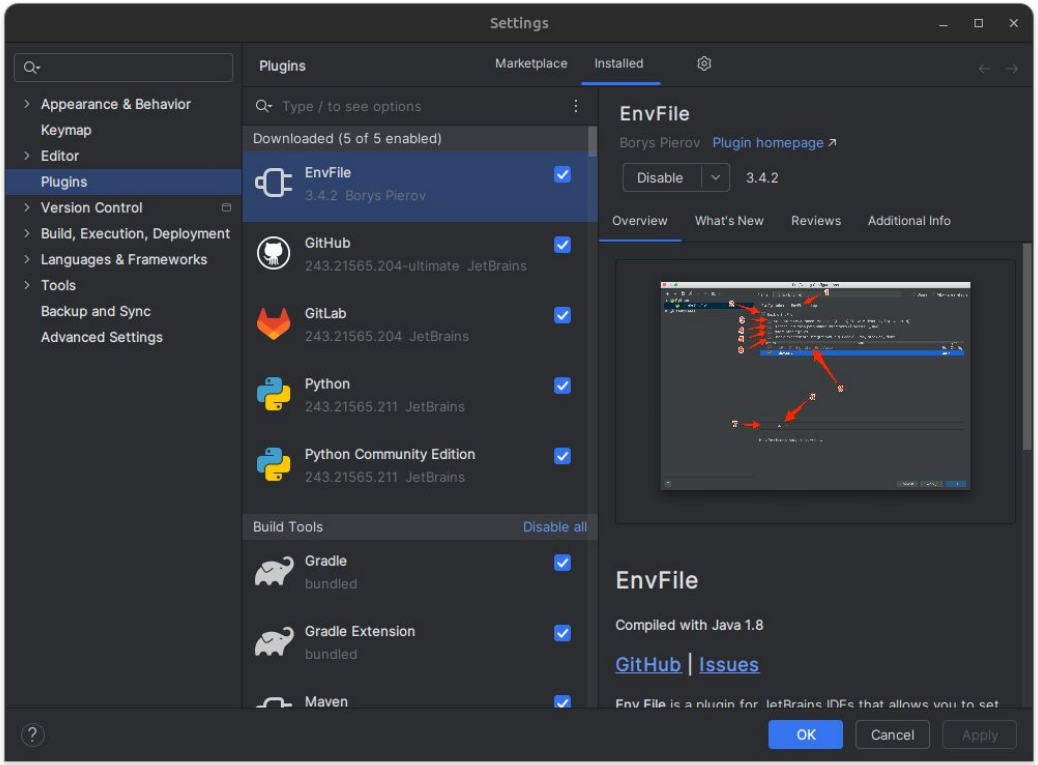
@DynamicPropertiesSource

```
@SpringBootTest
@Testcontainers
public class DatabaseTest {

                                                                                    @Container
              static    PostgreSQLContainer<?>    postgres    =    new    PostgreSQLContainer<>("postgres:11")
                                                                                .withDatabaseName("prop")
                                                                                .withUsername("postgres")
                                                                                    .withPassword("pass")
      .withExposedPorts(5432);

                                                                                  @DynamicPropertySource
                      static      void      registerPgProperties(DynamicPropertyRegistry      registry)      {
                                                                              registry.add("spring.datasource.url",
                () ->  String.format("jdbc:postgresql://localhost:%d/prop",  postgres.getFirstMappedPort()));
                                        registry.add("spring.datasource.username",     ()     ->     "postgres");
                                           registry.add("spring.datasource.password",      ()      ->      "pass");
    }

                                     //          tests          are          same          as          before
}
```

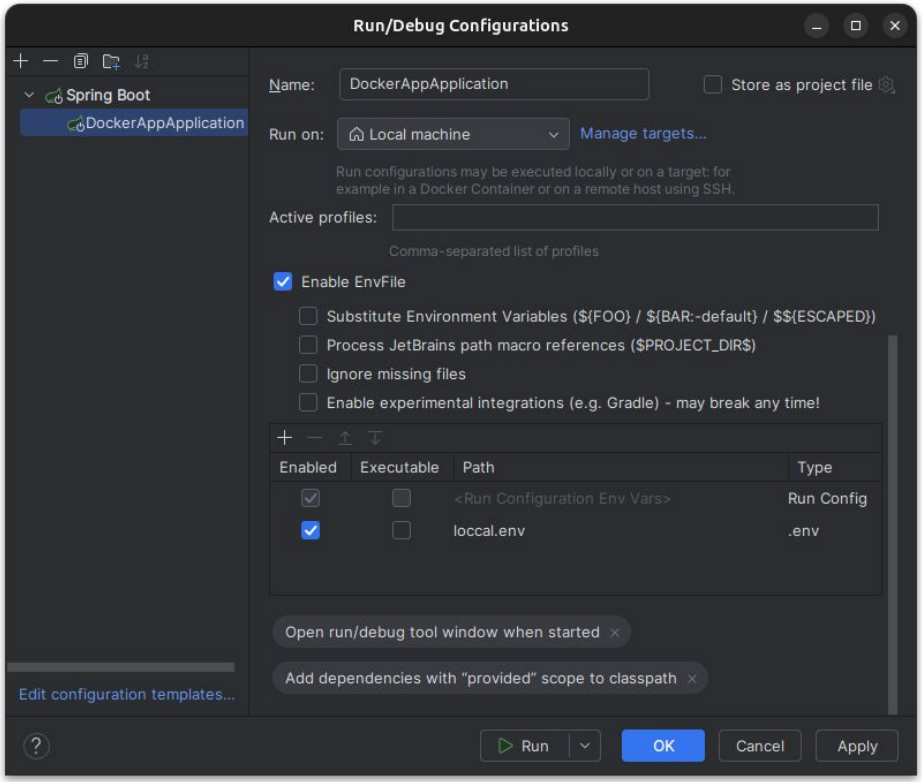
EnvFile

- EnvFile é um plugin para JetBrains IDEs que te permite atribuir variáveis de ambiente a partir de suas Run Configurations de um ou múltiplos arquivos.
- O plugin lê o(s) arquivo(s) `.env` configurados no Run Configurations relacionando o nome da propriedade ao nome da variável de ambiente e o valor da propriedade ao valor da variável de ambiente e inclui na subida da aplicação.

EnvFile


Settings
Plugins
Marketplace
Installed
Appearance & Behavior
Keymap
Editor
Plugins
Version Control
Build, Execution, Deployment
Languages & Frameworks
Tools
Backup and Sync
Advanced Settings
Type / to see options
Downloaded (5 of 5 enabled)
EnvFile
3.4.2 Borys Pierov
GitHub
243.21565.204-ultimate JetBrains
GitLab
243.21565.204 JetBrains
Python
243.21565.211 JetBrains
Python Community Edition
243.21565.211 JetBrains
Build Tools
Disable all
Gradle
bundled
Gradle Extension
bundled
Maven
EnvFile
Borys Pierov
Plugin homepage
Disable
3.4.2
Overview
What's New
Reviews
Additional Info
EnvFile
Compiled with Java 1.8
GitHub | Issues
OK
Cancel
Apply

EnvFile


Run/Debug Configurations
Spring Boot
DockerAppApplication
Edit configuration templates...
Name:
DockerAppApplication
Store as project file
Run on:
Local machine
Manage targets...
Run configurations may be executed locally or on a target: for example in a Docker Container or on a remote host using SSH.
Active profiles:
Comma-separated list of profiles
Enable EnvFile
Substitute Environment Variables (${FOO} / ${BAR:-default} / $${ESCAPED})
Process JetBrains path macro references ($PROJECT_DIR$)
Ignore missing files
Enable experimental integrations (e.g. Gradle) - may break any time!
Enabled
Executable
Path
Type
<Run Configuration Env Vars>
Run Config
loccal.env
.env
Open run/debug tool window when started
Add dependencies with "provided" scope to classpath
Run
OK
Cancel
Apply

Cases

- Na próxima apresentações sobre Spring Profile, teremos cases para reforçarmos esses conhecimentos obtidos nessa apresentação.
- Teremos os seguintes cases:
  - Case - Múltiplos profiles
  - Case - SQS Authentication
  - Case - Propriedades Dinâmicas

Q&A

- **E necessario manter propriedades iguais entre profiles (application.yaml e application-local.yaml)?**

Q&A

- **E necessario manter propriedades iguais entre profiles (application.yaml e application-local.yaml)?**

  Não é necessário manter as propriedades que forem iguais entre os profiles.

  Por exemplo: Se as configurações de API (contexto, porta e etc) forem as mesmas, ao manter no profile **default**, já vai poder ser reutilizado/herdado utilizado ao utilizar um profile **local**

Q&A

- **Se as propriedades forem iguais para eu rodar local, eu preciso manter o application.yaml e o application-local.yaml?**

Q&A

- **Se as propriedades forem iguais para eu rodar local, eu preciso manter o application.yaml e o application-local.yaml?**

  Não é necessário manter o profile **local**, porque o uso o **EnvFile** pode mudar os valores da variáveis locais do profile **default**

FIM