Profiles

A profile is a named, logical group of bean definitions to be registered with the container only if the given profile is active.

Profiles are useful for running application for different environment e.g. development, test, production etc. We can select/deselect different beans in different environment.

@Profile Annotation

This annotation indicates that the target component is eligible for registration when one or more profiles (specified by this annotation) are active.

Definition of Profile

```
@Target({ElementType.TYPE, ElementType.METHOD})

@Retention(RetentionPolicy.RUNTIME)

@Documented

@Conditional(ProfileCondition.class)

public @interface Profile {

    String[] value();

}
```

The value() attribute specifies the set of profiles for which the annotated component should be registered.

How to Use @Profile annotation?

The @Profile annotation may be used in any of the following ways:

On @Configuration classes

```
@Configuration

class CommonConfiguration {

}


@Configuration

@Profile("local")

class LocalConfiguration {


}


@Configuration

@Profile("dev")

class DevConfiguration {

}


@Configuration

@Profile("prod")

class ProdConfiguration {


}
```

The configurations/beans without profile annotation will be loaded for all profiles or if no profile is active.

On @Bean methods

```
@Configuration
@Profile("dev")
class DevConfiguration {

  @Profile("qa-tax-service")
  @Bean
  public WebClient taxServiceClient(){
      return ....
  }
}

@Configuration
@Profile("prod")
class ProdConfiguration {

  @Profile("uat-tax-service")
  @Bean
  public WebClient taxServiceClient(){
      return ....
  }
}
```

How to activate profile(s)?

We can activate profile(s) by one of the following ways:

By using the as a JVM system property having

key: spring.profiles.active

value: comma separated profile names.

Activating profiles programmatically via ConfigurableEnvironment.setActiveProfiles(java.lang.String...)

Naming a profile as default

Naming a Profile as "default" , has special meanings, i.e. if we don't activate any profile during startup, the "default" profile will be loaded along with the beans which don't have any profile associations. If we activate other profile, "default" will not be loaded. We can change "default" profile name by using context.getEnvironment().setDefaultProfiles() or by by using the spring.profiles.default property.

Example

Following example creates two profiles 'dev' (to run in a server environment) and 'local' (to run in a local machine)

Model

```
public class Customer {
 private String name;

  public Customer(String name) {
      this.name = name;
  }


  @Override
  public String toString() {
      return "Customer{" +
              "name='" + name + '\'' +
              '}';
  }
}
```

Data access beans

package Spring - Profiles

A profile is a named, logical group of bean definitions to be registered with the container only if the given profile is active.

Profiles are useful for running application for different environment e.g. development, test, production etc. We can select/deselect different beans in different environment.

@Profile Annotation

This annotation indicates that the target component is eligible for registration when one or more profiles (specified by this annotation) are active.

Definition of Profile

(Version: spring-framework 5.3.7)

```
package org.springframework.context.annotation;

  ........

@Target({ElementType.TYPE, ElementType.METHOD})

@Retention(RetentionPolicy.RUNTIME)

@Documented

@Conditional(ProfileCondition.class)

public @interface Profile {

    String[] value();

}
```

The value() attribute specifies the set of profiles for which the annotated component should be registered.

How to Use @Profile annotation?

The @Profile annotation may be used in any of the following ways:

On @Configuration classes

```
@Configuration

class CommonConfiguration {

}


@Configuration

@Profile("dev")

class LocalConfiguration {
}

@Configuration
@Profile("dev")
class DevConfiguration {
}

@Configuration
@Profile("prod")
class ProdConfiguration {

}
```

The configurations/beans without profile annotation will be loaded for all profiles or if no profile is active.

On @Bean methods

```
@Configuration
@Profile("dev")
class DevConfiguration {

  @Profile("qa-tax-service")
  @Bean
  public WebClient taxServiceClient(){
      return ....
  }
}

@Configuration
@Profile("prod")
class ProdConfiguration {

  @Profile("uat-tax-service")
  @Bean
  public WebClient taxServiceClient(){
      return ....
  }
}
```

How to activate profile(s)?

We can activate profile(s) by one of the following ways:

By using the as a JVM system property having

key: spring.profiles.active

value: comma separated profile names.

Activating profiles programmatically via ConfigurableEnvironment.setActiveProfiles(java.lang.String...)

Naming a profile as default

Naming a Profile as "default" , has special meanings, i.e. if we don't activate any profile during startup, the "default" profile will be loaded along with the beans which don't have any profile associations. If we activate other profile, "default" will not be loaded. We can change "default" profile name by using context.getEnvironment().setDefaultProfiles() or by by using the spring.profiles.default property.

Example

Following example creates two profiles 'dev' (to run in a server environment) and 'local' (to run in a local machine)

Model

```
public class Customer {
 private String name;

  public Customer(String name) {
      this.name = name;
  }

  @Override
  public String toString() {
      return "Customer{" +
              "name='" + name + '\'' +
              '}';
  }
}
```

Data access beans

```
public interface CustomerDao {
Customer getCustomer(String id);
}
@Repository
public class InMemoryCustomerDao implements CustomerDao{
  @Override
  public Customer getCustomer(String id) {
      return loadCustomerById(id);
  }

  private Customer loadCustomerById(String id) {
      return new Customer("in-memory-customer, id: "+id);
  }
}

@Repository
public class JpaCustomerDao implements CustomerDao{
  @Override
  public Customer getCustomer(String id) {
      return loadCustomerById(id);
  }
  private Customer loadCustomerById(String id) {
      return new Customer("db-loaded-customer, id: "+id);
  }
}
```

Service beans

```
public interface OrderService {
  void placeOrder(Customer customer, String orderDetails);
}

@Service
@Profile("local")
public class InMemoryOrderService implements OrderService {
  @Override
  public void placeOrder(Customer customer, String orderDetails) {
      System.out.println("InMemoryOrderService: order placed by "+customer+ " details: 
"+orderDetails);
  }
}

@Service
@Profile("dev")
public class OrderServiceSimulator implements OrderService{
  @Override
  public void placeOrder(Customer customer, String orderDetails) {
      System.out.println("OrderServiceSimulator: order placed by "+customer+ " details: 
"+orderDetails);
  }
}
```

Client bean

```
@Component
public class OrderClient {
 private CustomerDao customerDao;
 private OrderService orderService;

  public OrderClient(CustomerDao customerDao, OrderService orderService) {
      this.customerDao = customerDao;
      this.orderService = orderService;
  }

  public void placeOrder(String customerId){
      Customer customer =  customerDao.getCustomer(customerId);
      orderService.placeOrder(customer, "Convertible Thunder");
  }
}
```

Spring Java Config

```
@Configuration

public class DataAccessConfig {


  @Bean

  @Profile("local")

  public CustomerDao inMemoryCustomerDao(){

      return new InMemoryCustomerDao();

  }


  @Bean

  @Profile("dev")

  public CustomerDao japCustomerDao(){

      return new JpaCustomerDao();

  }

}
```

Following class has main method as well, which selects profiles programmatically.

```
@Configuration

@ComponentScan({"com.piseth.java.school.example.service",
"com.piseth.java.school.example.app"})

@Import(DataAccessConfig.class)

public class AppConfig {
  public static void main(String[] args) {
      runApp("local");
      System.out.println("---------");
      runApp( "dev");
  }

  private static void runApp(String profileName) {
      AnnotationConfigApplicationContext context =
              new AnnotationConfigApplicationContext();
      ConfigurableEnvironment env = context.getEnvironment();
      env.setActiveProfiles(profileName);
      context.register(AppConfig.class);
      context.refresh();

      OrderClient orderClient = context.getBean(OrderClient.class);
      orderClient.placeOrder("customer-1");
  }
}
```

Output

```
InMemoryOrderService: order placed by Customer{name='in-memory-customer, id:
customer-1'} details: Convertible Thunder
---------
OrderServiceSimulator: order placed by Customer{name='db-loaded-customer, id:
customer-1'} details: Convertible Thunder.example.dao;
```

```
public interface CustomerDao {
Customer getCustomer(String id);
}

@Repository
public class InMemoryCustomerDao implements CustomerDao{
  @Override
  public Customer getCustomer(String id) {
      return loadCustomerById(id);
  }

  private Customer loadCustomerById(String id) {
      return new Customer("in-memory-customer, id: "+id);
  }
}

@Repository
public class JpaCustomerDao implements CustomerDao{
  @Override
  public Customer getCustomer(String id) {
      return loadCustomerById(id);
  }
  private Customer loadCustomerById(String id) {
      return new Customer("db-loaded-customer, id: "+id);
  }
}
```

Service beans

```
public interface OrderService {
  void placeOrder(Customer customer, String orderDetails);
}

@Service
@Profile("local")
public class InMemoryOrderService implements OrderService {
  @Override
  public void placeOrder(Customer customer, String orderDetails) {
      System.out.println("InMemoryOrderService: order placed by "+customer+ " details: 
"+orderDetails);
  }
}
import org.springframework.context.annotation.Profile;

@Service
@Profile("dev")
```

```
public class OrderServiceSimulator implements OrderService{

  @Override
  public void placeOrder(Customer customer, String orderDetails) {
      System.out.println("OrderServiceSimulator: order placed by "+customer+ " details:
"+orderDetails);
  }
}
```

Client bean

```
@Component
public class OrderClient {
 private CustomerDao customerDao;
 private OrderService orderService;

  public OrderClient(CustomerDao customerDao, OrderService orderService) {
      this.customerDao = customerDao;
      this.orderService = orderService;
  }

  public void placeOrder(String customerId){
      Customer customer = customerDao.getCustomer(customerId);
      orderService.placeOrder(customer, "Convertible Thunder");
  }
}
```

Spring Java Config

```
@Configuration
public class DataAccessConfig {

  @Bean
  @Profile("local")
  public CustomerDao inMemoryCustomerDao(){
      return new InMemoryCustomerDao();
  }

  @Bean
  @Profile("dev")
  public CustomerDao japCustomerDao(){
      return new JpaCustomerDao();
  }
}
```

Following class has main method as well, which selects profiles programmatically.

```
@Configuration
@ComponentScan({"com.piseth.java.school.example.service",
"com.piseth.java.school.example.app"})
```

```
@Import(DataAccessConfig.class)
public class AppConfig {


  public static void main(String[] args) {
      runApp("local");
      System.out.println("---------");
      runApp( "dev");
  }


  private static void runApp(String profileName) {
      AnnotationConfigApplicationContext context =
              new AnnotationConfigApplicationContext();
      ConfigurableEnvironment env = context.getEnvironment();
      env.setActiveProfiles(profileName);
      context.register(AppConfig.class);
      context.refresh();


      OrderClient orderClient = context.getBean(OrderClient.class);
      orderClient.placeOrder("customer-1");
  }
}
```

Output

```
InMemoryOrderService: order placed by Customer{name='in-memory-customer, id:
customer-1'} details: Convertible Thunder
```

```
---------

OrderServiceSimulator: order placed by Customer{name='db-loaded-customer, id:
customer-1'} details: Convertible Thunder
```

Accessing Environment Properties

What is Environment?

org.springframework.core.env.Environment is an interface representing the environment in which the current application is running.

With Environment instance we can access the properties loaded for the application. Let's understand how can we do that.

Definition of Environment

```
package org.springframework.core.env;

  ........

public interface Environment extends PropertyResolver {

   String[] getActiveProfiles();

   String[] getDefaultProfiles();

   @Deprecated

   boolean acceptsProfiles(String... profiles);

   boolean acceptsProfiles(Profiles profiles);

}
```

As seen above Environment extends PropertyResolver.

Definition of PropertyResolver

```
package org.springframework.core.env;

  ........

public interface PropertyResolver {

   boolean containsProperty(String key);

   @Nullable

   String getProperty(String key);

   String getProperty(String key, String defaultValue);

   @Nullable

   <T> T getProperty(String key, Class<T> targetType);

   <T> T getProperty(String key, Class<T> targetType, T defaultValue);

   String getRequiredProperty(String key) throws IllegalStateException;

   <T> T getRequiredProperty(String key, Class<T> targetType)

         throws IllegalStateException;

   String resolvePlaceholders(String text);

   String resolveRequiredPlaceholders(String text) throws IllegalArgumentException;

}
```

Spring attempts to unify all name/value property pairs access into org.springframework.core.env.Environment.

The properties source can be java.util.Properties, loaded from a file or Java system/env properties or java.util.Map.

If we are in the Servlet container environment, the source can be javax.servlet.ServletContext or javax.servlet.ServletConfig.

ConfigurableEnvironment interface

This interface extends Environment and ConfigurablePropertyResolver interfaces. It provides facilities for setting active and default profiles and accessing underlying property sources.

Definition of ConfigurableEnvironment

```
package org.springframework.core.env;

  ........

public interface ConfigurableEnvironment extends Environment {

   void setActiveProfiles(String... profiles);

   void addActiveProfile(String profile);

   void setDefaultProfiles(String... profiles);

   MutablePropertySources getPropertySources();

   Map<String, Object> getSystemProperties();

   Map<String, Object> getSystemEnvironment();

   void merge(ConfigurableEnvironment parent);

}
```

Definition of ConfigurablePropertyResolver

```
package org.springframework.core.env;

  ........

public interface ConfigurablePropertyResolver extends PropertyResolver {
    ConfigurableConversionService getConversionService();

    void setConversionService(ConfigurableConversionService conversionService);

    void setPlaceholderPrefix(String placeholderPrefix);

    void setPlaceholderSuffix(String placeholderSuffix);

    void setValueSeparator(@Nullable String valueSeparator);

    void setIgnoreUnresolvableNestedPlaceholders(
            boolean ignoreUnresolvableNestedPlaceholders);

    void setRequiredProperties(String... requiredProperties);

    void validateRequiredProperties() throws MissingRequiredPropertiesException;

}
```

We can get an instance of ConfigurableEnvironment via ConfigurableApplicationContext#getEnvironment() (as we saw in the last tutorial). ConfigurableApplicationContext is a sub interface of org.springframework.context.ApplicationContext.

Accessing properties with Environment

Using methods of org.springframework.core.env.Environment directly is one way to access properties in our applications.

```
public class DefaultSystemSourcesExample {

  public static void main (String[] args) {

      AnnotationConfigApplicationContext context =

                    new AnnotationConfigApplicationContext();

      ConfigurableEnvironment env = context.getEnvironment();

      printSources(env);
      System.out.println("-- System properties --");
      printMap(env.getSystemProperties());
      System.out.println("-- System Env properties --");
      printMap(env.getSystemEnvironment());
  }

  private static void printSources (ConfigurableEnvironment env) {
      System.out.println("-- property sources --");
      for (PropertySource<?> propertySource : env.getPropertySources()) {
          System.out.println("name =  " + propertySource.getName() + "\nsource = " +
propertySource
                              .getSource().getClass()+"\n");
      }
  }

  private static void printMap (Map<?, ?> map) {
      map.entrySet()
         .stream().limit(15)
         .forEach(e -> System.out.println(e.getKey() + " = " + e.getValue()));
      System.out.println("-------------");

  }
}
```

Output

```
-- property sources --
name =  systemProperties
source = class java.util.Properties


name =  systemEnvironment
source = class java.util.Collections$UnmodifiableMap


-- System properties --
exec.cleanupDaemonThreads = false
java.runtime.name = Java(TM) SE Runtime Environment
exec.mainClass = com.piseth.java.school.example.DefaultSystemSourcesExample
sun.boot.library.path = D:\programs\java\jdk1.8.0_151\jre\bin
java.vm.version = 25.151-b12
java.vm.vendor = Oracle Corporation
java.vendor.url = http://java.oracle.com/
guice.disable.misplaced.annotation.check = true
path.separator = ;
java.vm.name = Java HotSpot(TM) 64-Bit Server VM
file.encoding.pkg = sun.io
user.country = US
user.script =
sun.java.launcher = SUN_STANDARD
-------------
```

```
-- System Env properties --
example-site-projects = D:\project\example-sentence-site
USERDOMAIN_ROAMINGPROFILE = DESKTOP-0E87003
GIT_HOME = D:\programs\Git
PROCESSOR_LEVEL = 6
SESSIONNAME = Console
ALLUSERSPROFILE = C:\ProgramData
PROCESSOR_ARCHITECTURE = AMD64
intellijPath = D:\programs\JetBrains\ideaIC-2021.1.1.win\bin
jvmConfig = \.mvn\jvm.config
PSModulePath = C:\Program Files\WindowsPowerShell\Modules;C:\WINDOWS\system32\WindowsPowerShell\v1.0\Modules
SystemDrive = C:
=ExitCode = 00000001
-------------
```

As seen in above output, there are two property sources by default, System properties and System Environmental properties. We can add our own property sources as well (next tutorials).

By default, system properties have precedence over environment variables, so if the foo property happens to be set in both places during a call to env.getProperty("foo"), the system property value will 'win' and be returned preferentially over the environment variable. Note that property values will not get merged but rather completely overridden by a preceding entry.

Adding New Property Source to Environment

In this tutorial we will see how to add our own properties to Spring Environment programmatically.

Example

Adding user defined properties

src/main/resources/app.properties

```
contact-person=Piseth Ing
```

Adding new Property source and accessing in Spring

```
public class UserPropertySourceExample {
  public static void main(String[] args) throws IOException {
      AnnotationConfigApplicationContext context =
              new AnnotationConfigApplicationContext();
      ConfigurableEnvironment env = context.getEnvironment();
      env.getPropertySources().addLast(new ResourcePropertySource(new
ClassPathResource("app.properties")));
      printSources(env);
      String contactPerson = env.getProperty("contact-person");
      System.out.println("-- accessing  app.properties --");
      System.out.println("contact-person: " + contactPerson);
  }

  private static void printSources(ConfigurableEnvironment env) {
    System.out.println("-- property sources --");
    for (PropertySource<?> propertySource : env.getPropertySources()) {
        System.out.println("name =  " + propertySource.getName() + "\nsource = " +
propertySource
                .getSource().getClass() + "\n");
    }
  }
}
```

Output

```
-- property sources --
name =  systemProperties
source = class java.util.Properties

name =  systemEnvironment
source = class java.util.Collections$UnmodifiableMap

name =  class path resource [app.properties]
source = class java.util.Properties

-- accessing  app.properties --
contact-person: Piseth Ing
```

Adding user properties by using @PropertySource

In this tutorial we will see how to add user properties to Spring Environment using @PropertySource.

@PropertySource annotation is used on @Configuration classes.

Definition of PropertySource

```
package org.springframework.context.annotation;

 ........

@Target(ElementType.TYPE)

@Retention(RetentionPolicy.RUNTIME)

@Documented

@Repeatable(PropertySources.class)

public @interface PropertySource {

    String name() default ""; 1

    String[] value(); 2

    boolean ignoreResourceNotFound() default false; 3

    String encoding() default ""; 4

    Class<? extends PropertySourceFactory> factory() default

            PropertySourceFactory.class; 5

}
```

1 Indicates the name of this property source. If omitted, a name will be generated based on the underlying resource.

2 Indicates the resource location(s) of the properties file to be loaded.

Examples: "classpath:/com/piseth/app.properties" or "file:/path/to/file.xml".

Wildcards (e.g. **/*.properties) are not permitted.

${...} placeholders will be resolved against any/all property sources already registered with the Environment.

3 Indicates if a failure to find the property resource should be ignored. Default is false. Should be set to true if the properties file is completely optional.

4 A specific character encoding for the given resources, e.g. "UTF-8"

5 Specifies a custom PropertySourceFactory (Strategy interface for creating resource-based PropertySource wrappers). By default DefaultPropertySourceFactory is used.

Example

User property file

src/main/resources/app.properties

```
contact-person=Piseth Ing
```

Using @PropertySource

```
@Configuration
@PropertySource("classpath:app.properties")
public class UserPropertySourceExample {
  public static void main(String[] args) {
      AnnotationConfigApplicationContext context =
              new AnnotationConfigApplicationContext(UserPropertySourceExample.class);
      ConfigurableEnvironment env = context.getEnvironment();
      System.out.println("property sources:" + env.getPropertySources());

      String contactPerson = env.getProperty("contact-person");

      System.out.println("contact-person: " + contactPerson);

  }

}
```

Output

```
property sources:[PropertiesPropertySource {name='systemProperties'},
SystemEnvironmentPropertySource {name='systemEnvironment'},
ResourcePropertySource {name='class path resource [app.properties]'}]

contact-person: Piseth Ing
```

Injecting Environment to access properties in beans

Instead of getting Environment instance by using ConfigurableApplicationContext#getEnvironment() we can directly inject it as a bean.

Example

Autowiring Environment

```
@Component
public class MyBean {
  @Autowired
  private Environment environment;

  @PostConstruct
  public void postInit() {
      System.out.println("-- accessing system properties --");
      String tempDir = environment.getProperty("java.io.tmpdir");
      System.out.println("System tempDir: "+tempDir);
      System.out.println("-- accessing user properties --");
      String contactPerson = environment.getProperty("contact-person");
      System.out.println("contact-person: " + contactPerson);
  }
}
```

Property file

src/main/resources/app.properties

```
contact-person=Piseth Ing
```

Main class

```
@Configuration
@PropertySource("classpath:app.properties")
@ComponentScan
public class UserPropertySourceExample {
  public static void main(String[] args) {
      new AnnotationConfigApplicationContext(UserPropertySourceExample.class);
  }
}
```

Output

```
-- accessing system properties --
System tempDir: C:\Users\joe\AppData\Local\Temp\
-- accessing user properties --
contact-person: Piseth Ing
```

Using @Value Annotation

In previous tutorials we saw different ways to access properties provided by Environment. In this tutorial we will see how to use @Value annotation to access individual properties without using Environment.

Definition of Value

```
package org.springframework.beans.factory.annotation;

  ........

@Target({ElementType.FIELD, ElementType.METHOD, ElementType.PARAMETER,
ElementType.ANNOTATION_TYPE})

@Retention(RetentionPolicy.RUNTIME)

@Documented

public @interface Value {

    String value();

}
```

The actual value expression such as #{systemProperties.myProp} or property placeholder such as ${my.app.myProp}.

Example

External Property File

src/main/resources/app.properties

```
contact-person=Piseth Ing
```

Using @Value

```
@Component

public class MyBean {

  @Value("${java.io.tmpdir:/temp}")
  private String tempDir;

  @Value("${contact-person:Joe}")

  private String contactPerson;


  @PostConstruct

  public void postInit() {

      System.out.println("System tempDir: " + tempDir);

      System.out.println("contact-person: " + contactPerson);

  }

}
```

It is possible to provide a default value after colon as seen above ('Joe' is the default value for contact person). So if a property (contact-person in our example) cannot be found the default value will be use at the injection point.

Java Config and main method

```
@Configuration

@PropertySource("classpath:app.properties")

@ComponentScan

public class UserPropertySourceExample {


  @Bean

  public static PropertySourcesPlaceholderConfigurer propertyPlaceholderConfigurer() {

      return new PropertySourcesPlaceholderConfigurer();

  }
  public static void main(String[] args) {
      new AnnotationConfigApplicationContext(UserPropertySourceExample.class);
  }
}
```

Output

```
System tempDir: C:\Users\joe\AppData\Local\Temp\
contact-person: Piseth Ing
```

As seen in above class, we also have to register PropertySourcesPlaceholderConfigurer as a bean for @Value annotation to work.

Using Spring Expression Language with @Value Annotation

This example shows how to use spring expression language in value element of @Value.

Example

Using expression language

```
@Component

public class MyBean {

  @Value("#{systemProperties['user.home']}")

  private String userHome;


  @Value("#{T(java.lang.Math).random()*1000}")

  private int randomNumber;


  @PostConstruct

  public void postInit() {

      System.out.println("System userHome: " + userHome);

      System.out.println("Random number: " + randomNumber);

  }

}
```

Main class

```
@Configuration

@ComponentScan

public class ExampleMain {
  public static void main(String[] args) {

      new AnnotationConfigApplicationContext(ExampleMain.class);

  }

}
```

Output

```
System userHome: C:\Users\joe

Random number: 450
```