Auditing

Basics

Spring Data provides sophisticated support to transparently keep track of who created or changed an entity and when the change happened.To benefit from that functionality, you have to equip your entity classes with auditing metadata that can be defined either using annotations or by implementing an interface. Additionally, auditing has to be enabled either through Annotation configuration or XML configuration to register the required infrastructure components. Please refer to the store-specific section for configuration samples.

NOTE

Applications that only track creation and modification dates are not required do make their entities implement `AuditorAware`.

Annotation-based Auditing Metadata

We provide `@CreatedBy` and `@LastModifiedBy` to capture the user who created or modified the entity as well as `@CreatedDate` and `@LastModifiedDate` to capture when the change happened.

An audited entity

```java
class Customer {

  @CreatedBy
  private User user;

  @CreatedDate
  private Instant createdDate;

  // … further properties omitted
}
```

As you can see, the annotations can be applied selectively, depending on which information you want to capture. The annotations, indicating to capture when changes are made, can be used on properties of type JDK8 date and time types, `long`, `Long`, and legacy Java `Date` and `Calendar`.

Auditing metadata does not necessarily need to live in the root level entity but can be added to an embedded one (depending on the actual store in use), as shown in the snippet below.

Audit metadata in embedded entity

```java
class Customer {

  private AuditMetadata auditingMetadata;

  // … further properties omitted
}

class AuditMetadata {

  @CreatedBy
  private User user;

  @CreatedDate
  private Instant createdDate;

}
```

Interface-based Auditing Metadata

In case you do not want to use annotations to define auditing metadata, you can let your domain class implement the `Auditable` interface. It exposes setter methods for all of the auditing properties.

`AuditorAware`

In case you use either `@CreatedBy` or `@LastModifiedBy`, the auditing infrastructure somehow needs to become aware of the current principal. To do so, we provide an `AuditorAware<T>` SPI interface that you have to implement to tell the infrastructure who the current user or system interacting with the application is. The generic type `T` defines what type the properties annotated with `@CreatedBy` or `@LastModifiedBy` have to be.

The following example shows an implementation of the interface that uses Spring Security's `Authentication` object:

Implementation of `AuditorAware` based on Spring Security

```java
class SpringSecurityAuditorAware implements AuditorAware<User> {

  @Override
  public Optional<User> getCurrentAuditor() {

    return Optional.ofNullable(SecurityContextHolder.getContext())
            .map(SecurityContext::getAuthentication)
            .filter(Authentication::isAuthenticated)
            .map(Authentication::getPrincipal)
            .map(User.class::cast);
  }
}
```

The implementation accesses the `Authentication` object provided by Spring Security and looks up the custom `UserDetails` instance that you have created in your `UserDetailsService` implementation. We assume here that you are exposing the domain user through the `UserDetails` implementation but that, based on the `Authentication` found, you could also look it up from anywhere.

`ReactiveAuditorAware`

When using reactive infrastructure you might want to make use of contextual information to provide `@CreatedBy` or `@LastModifiedBy` information. We provide an `ReactiveAuditorAware<T>` SPI interface that you have to implement to tell the infrastructure who the current user or system interacting with the application is. The generic type `T` defines what type the properties annotated with `@CreatedBy` or `@LastModifiedBy` have to be.

The following example shows an implementation of the interface that uses reactive Spring Security's `Authentication` object:

Implementation of `ReactiveAuditorAware` based on Spring Security

```java
class SpringSecurityAuditorAware implements ReactiveAuditorAware<User> {

  @Override
  public Mono<User> getCurrentAuditor() {

    return ReactiveSecurityContextHolder.getContext()
                .map(SecurityContext::getAuthentication)
                .filter(Authentication::isAuthenticated)
                .map(Authentication::getPrincipal)
                .map(User.class::cast);
  }
}
```

The implementation accesses the `Authentication` object provided by Spring Security and looks up the custom `UserDetails` instance that you have created in your `UserDetailsService` implementation. We assume here that you are exposing the domain user through the `UserDetails` implementation but that, based on the `Authentication` found, you could also look it up from anywhere.

There is also a convenience base class, `AbstractAuditable`, which you can extend to avoid the need to manually implement the interface methods. Doing so increases the coupling of your domain classes to Spring Data, which might be something you want to avoid. Usually, the annotation-based way of defining auditing metadata is preferred as it is less invasive and more flexible.

General Auditing Configuration

Spring Data JPA ships with an entity listener that can be used to trigger the capturing of auditing information. First, you must register the `AuditingEntityListener` to be used for all entities in your persistence contexts inside your `orm.xml` file, as shown in the following example:

Example 1. Auditing configuration orm.xml

```xml
<persistence-unit-metadata>
  <persistence-unit-defaults>
    <entity-listeners>
      <entity-listener class="….data.jpa.domain.support.AuditingEntityListener" />
    </entity-listeners>
  </persistence-unit-defaults>
</persistence-unit-metadata>
```

You can also enable the `AuditingEntityListener` on a per-entity basis by using the `@EntityListeners` annotation, as follows:

```java
@Entity
@EntityListeners(AuditingEntityListener.class)
public class MyEntity {

}
```

NOTE

The auditing feature requires `spring-aspects.jar` to be on the classpath.

With `orm.xml` suitably modified and `spring-aspects.jar` on the classpath, activating auditing functionality is a matter of adding the Spring Data JPA `auditing` namespace element to your configuration, as follows:

Example 2. Activating auditing using XML configuration

```xml
<jpa:auditing auditor-aware-ref="yourAuditorAwareBean" />
```

As of Spring Data JPA 1.5, you can enable auditing by annotating a configuration class with the `@EnableJpaAuditing` annotation. You must still modify the `orm.xml` file and have `spring-aspects.jar` on the classpath. The following example shows how to use the `@EnableJpaAuditing` annotation:

Example 3. Activating auditing with Java configuration

```java
@Configuration
@EnableJpaAuditing
class Config {

  @Bean
  public AuditorAware<AuditableUser> auditorProvider() {
    return new AuditorAwareImpl();
  }
}
```

If you expose a bean of type `AuditorAware` to the `ApplicationContext`, the auditing infrastructure automatically picks it up and uses it to determine the current user to be set on domain types. If you have multiple implementations registered in the `ApplicationContext`, you can select the one to be used by explicitly setting the `auditorAwareRef` attribute of `@EnableJpaAuditing`.

Auditing
Basics
Annotation-based Auditing Metadata
Interface-based Auditing Metadata
AuditorAware
ReactiveAuditorAware
General Auditing Configuration

Edit this Page
GitHub Project
Stack Overflow