Registration – Activate a New Account by Email | Technical Articles

Abstract

1. Overview This article continues our ongoing Registration with Spring Security series by finishing the missing piece of the registration process - verifying

Table of Contents

Navigation

Technical Articles [http://inprogress.baeldung.com/]

Home [http://inprogress.baeldung.com]

- Home [http://inprogress.baeldung.com/]
- http://inprogress.baeldung.com/?feed=rss2

Return to Content

Registration – Activate a New Account by Email

By

Elena [http://inprogress.baeldung.com/?author=10]

on
October 23, 2014

in

Spring Security [http://inprogress.baeldung.com/?cat=9]

1. Overview

This article continues our ongoing **Registration with Spring Security series** by finishing the missing piece of the registration process – **verifying the email to confirm the user registration**.

The registration confirmation mechanism forces the user to respond to a "***Confirm Registration***" email sent after successful registration to verify his email address and activate his account. The user does this by clicking a unique account activation link sent to him as part of the email message.

Following this logic, a newly registered user will not be able to log in until email/registration verification is completed.

2. A Verification Token

We will make use of a simple verification token as the key artifact through which a user is verified.

2.1. Adding a VerificationToken Entity to Our Model

The *VerificationToken* entity must meet the following criteria:

1. There will be one *VerificationToken* associated to a *User*. So, we need a one-to-one unidirectional association between the *VerificationToken* and the *User*.

2. It will be created after the user registration data is persisted.

3. It will expire in 24 hours following initial registration.

4. Its value should be unique and randomly generated.

Requirements 2 and 3 are part of the registration logic. The other two are implemented in a simple *VerificationToken* entity like the one in Example 2.1.:

Example 2.1.

```
@Entity
@Table
public class VerificationToken {
    private static final int EXPIRATION = 60 * 24;

    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    private Long id;

    @Column(name = "token")
    private String token;

    @OneToOne(targetEntity = User.class, fetch = FetchType.EAGER)
    @JoinColumn(name = "user_id")
    private User user;

    @Column(name = "expiry_date")
    private Date expiryDate;

    public VerificationToken() {
        super();
    }
    public VerificationToken(String token, User user) {
        super();
        this.token = token;
        this.user = user;
```

```
        this.expiryDate = calculateExpiryDate(EXPIRATION);
        this.verified = false;
    }
    private Date calculateExpiryDate(int expiryTimeInMinutes) {
        Calendar cal = Calendar.getInstance();
        cal.setTime(new Timestamp(cal.getTime().getTime()));
        cal.add(Calendar.MINUTE, expiryTimeInMinutes);
        return new Date(cal.getTime().getTime());
    }

    // standard getters and setters
}
```

2.2. Add an Enabled Flag to the User Entity

We will set the value of this flag depending on the result of the registration confirmation use case. Lets jus add the following field to our *User* entity for now:

```
@Column(name = "enabled")
private boolean enabled;
```

3. The Account Registration Phase

Lets add two additional pieces of business logic to the user registration use case:

1. Generating a *VerificationToken* for the user and persisting it.

2. Sending the account confirmation email message which includes a confirmation link with the *VerificationToken's* valueas a parameter.

3.1. Using Spring Event Handling to Create the Token and Send the Verification Email

These two additional pieces of logic should not be performed by the controller directly because they are "collateral" back-end tasks. The controller will publish a Spring *ApplicationEvent* to trigger the execution of these tasks. This is as simple as injecting an *ApplicationEventPublishe*r in the controller, and then using it to publish the registration completion. Example 3.1. shows this simple logic:

Example 3.1.

```
@Autowired
ApplicationEventPublisher
@RequestMapping(value = "/user/registration", method = RequestMethod.POST)
public ModelAndView registerUserAccount(@ModelAttribute("user") @Valid UserDto acc
      BindingResult result, WebRequest request, Errors errors) {
    User registered = new User();
    String appUrl = request.getContextPath();
    if (result.hasErrors()) {
       return new ModelAndView("registration", "user", accountDto);
    }
    registered = createUserAccount(accountDto);
    if (registered == null) {
        result.rejectValue("email", "message.regError");
    }
```

```
        eventPublisher.publishEvent(new OnRegistrationCompleteEvent(registered,
          request.getLocale(), appUrl));
        return new ModelAndView("successRegister", "user", accountDto);
    }
```

3.2. Spring Event Handler Implementation

The controller is using an *ApplicationEventPublisher* to start the *RegistrationListener* that will handle the verification token creation and confirmation email sending. So it needs to have access to the implementation of the following interfaces:

1. An **AplicationEvent** representing the completion of the user registration.

2. An **ApplicationListener** bean which will listen to the published event and proceed to do all the work.

The beans we will create are the *OnRegistrationCompleteEvent* , and the *RegistrationListener* shown Examples 3.2.1 – 3.2.2.

Example 3.2.1. – The *OnRegistrationCompleteEvent*

```
@SuppressWarnings("serial")
public class OnRegistrationCompleteEvent extends ApplicationEvent {
    private final String appUrl;
    private final Locale locale;
    private final User user;

    public OnRegistrationCompleteEvent(User user, Locale locale, String appUrl) {
        super(user);
        this.user = user;
        this.locale = locale;
        this.appUrl = appUrl;
    }

    // standard getters and setters
}
```

Example 3.2.2. - The RegistrationListener Responds to the *OnRegistrationCompleteEvent*

```
@Component
public class RegistrationListener implements ApplicationListener<OnRegistrationCom
    @Autowired
    private IUserService service;

    @Autowired
    private MessageSource messages;

    @Autowired
    private JavaMailSender mailSender;

    @Override
    public void onApplicationEvent(OnRegistrationCompleteEvent event) {
        this.confirmRegistration(event);
    }

    private void confirmRegistration(OnRegistrationCompleteEvent event) {
```

```
        User user = event.getUser();
        String token = UUID.randomUUID().toString();
        service.addVerificationToken(user, token);
        String recipientAddress = user.getEmail();
        String subject = "Registration Confirmation";
        String confirmationUrl = event.getAppUrl() + "/regitrationConfirm.html?tok
        String message = messages.getMessage("message.regSucc", null, event.getLoc
        SimpleMailMessage email = new SimpleMailMessage();
        email.setTo(recipientAddress);
        email.setSubject(subject);
        email.setText(message + " \r\n" + "http://localhost:8080" + confirmationUr
        mailSender.send(email);
    }
}
```

Here, the *confirmRegistration* method will receive the *OnRegistrationCompleteEvent*, extract all the necessary *User* information from it, create the verification token, persist it, and then send it as a parameter in the "Confirm Registration" link sent to the user.

3.3. Processing the Verification Token Parameter

When the user receives the "Confirm Registration" email, he will click on the attached link and fire a GET request. The controller will extract the value of the token parameter in the GET request and will use it to verify the user. Lets see this logic in Example 3.3.1.

Example 3.3.1. – RegistrationController Processing the Registration Confirmation Link

```
private IUserService service;

@Autowired
public RegistrationController(IUserService service){
    this.service = service
}
@RequestMapping(value = "/regitrationConfirm", method = RequestMethod.GET)
public String confirmRegistration(WebRequest request, Model model,
      @RequestParam("token") String token) {
    VerificationToken verificationToken = service.getVerificationToken(token);
    if (verificationToken == null) {
        model.addAttribute("message", messages.getMessage("auth.message.invalidTok
          null, request.getLocale()));
        return "redirect:/badUser.html?lang=" + request.getLocale().getLanguage();
    }
    User user = verificationToken.getUser();
    Calendar cal = Calendar.getInstance();
    if (user == null) {
        model.addAttribute("message", messages.getMessage("auth.message.invalidUse
          null, request.getLocale()));
        return "redirect:/badUser.html?lang=" + request.getLocale().getLanguage();
    }
    if ((verificationToken.getExpiryDate().getTime() - cal.getTime().getTime()) <=
        user.setEnabled(false);
    } else {
        user.setEnabled(true);
    }
```

```
    service.saveRegisteredUser(user);
    return "redirect:/login.html?lang=" + request.getLocale().getLanguage();
}
```

Notice that if there is no user associated with the *VerificationToken* or if the *VerificationToken* does not exist, the controller will return a *badUser.html* page with the corresponding error message (See Example 3.3.2.).

Example 3.3.2. – The badUser.html

```
<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c" %>
<%@ taglib prefix="sec" uri="http://www.springframework.org/security/tags"%>
<%@taglib uri="http://www.springframework.org/tags" prefix="spring"%>
<%@ taglib uri="http://java.sun.com/jsp/jstl/fmt" prefix="fmt"%>
<fmt:setBundle basename="messages" />
<%@ page session="true"%>
<html>
<head>
    <link href="<c:url value="/resources/bootstrap.css" />" rel="stylesheet">
    <title>Expired</title>
</head>
<body>
    <h1>${message}</h1>
    <br>
    <a href="<c:url value="/user/registration" />">
        <spring:message code="label.form.loginSignUp"></spring:message>
    </a>
</body>
</html>
```

If the token and user exist, the controller then proceeds to set the *User*‘s *enabled* field after checking if the *VerificationToken* has expired.

4. Adding Account Activation Checking to the Login Process

We need to add the following verification logic to My*UserDetailsService’s* **lo*adUserByUsername*** method:

- Make sure that the user is enabled before letting him log in.

Example 4.1. shows the simple *isEnabled()* check.

Example 4.1. – Checking the VerificationToken **in MyUserDetailsService**

```
private UserRepository userRepository;
@Autowired
private IUserService service;
@Autowired
private MessageSource messages;

@Autowired
public MyUserDetailsService(UserRepository repository) {
    this.userRepository = repository;
}
```

```
public UserDetails loadUserByUsername(String email) throws UsernameNotFoundExcepti
    boolean enabled = true;
    boolean accountNonExpired = true;
    boolean credentialsNonExpired = true;
    boolean accountNonLocked = true;
    try {
        User user = userRepository.findByEmail(email);
        if (user == null) {
            return new org.springframework.security.core.userdetails.User(" ", " "
                    true, true, true, getAuthorities(new Integer(1)));
        }
        if (!user.isEnabled()) {
            accountNonExpired = false;
            service.deleteUser(user);
            return new org.springframework.security.core.userdetails.User(" ", " "
              accountNonExpired, true, true, getAuthorities(new Integer(1)));
        }
        return new org.springframework.security.core.userdetails.User(user.getEmai
          user.getPassword().toLowerCase(),
          enabled, accountNonExpired, credentialsNonExpired, accountNonLocked,
          getAuthorities(user.getRole().getRole()));
    } catch (Exception e) {
        throw new RuntimeException(e);
    }
}
```

Notice that if the user is not enabled, the account is deleted and the method returns an *org.springframework.security.core.userdetails.User* with the *accountNonExpired* flag set to false. This will trigger a **SPRING_SECURITY_LAST_EXCEPTION** in the login process. This exception's String value is: '*User Account Has Expired*'.

Now, we need to modify our *login.html* page to show this and any other exception messages resulting from en email verification error. The error checking code we added to *login.html* is shown in Example 4.2.*:*

Example 4.2. – Adding Account Activation Error Checking to l*ogin.html*

```
<c:if test="${param.error != null}">
    <c:choose>
        <c:when test="${SPRING_SECURITY_LAST_EXCEPTION.message == 'User is disable
            <div class="alert alert-error">
                <spring:message code="auth.message.disabled"></spring:message>
            </div>
        </c:when>
        <c:when test="${SPRING_SECURITY_LAST_EXCEPTION.message == 'User account ha
            <div class="alert alert-error">
                <spring:message code="auth.message.expired"></spring:message>
            </div>
        </c:when>
        <c:otherwise>
            <div class="alert alert-error">
           <spring:message code="message.badCredentials"></spring:message>
              </div>
        </c:otherwise>
    </c:choose>
```

```
</c:if>
```

5. Adapting the Persistence Layer

We need to modify the API of the persistence layer by:

1. Creating a *VerificationTokenRepository*. For *User* and *VerificationToken* access.

2. Adding methods to the *IUserInterface* and its implementation for new CRUD operations needed.

Examples 5.1 – 5.3. show the new interfaces and implementation:

Example 5.1. – The *VerificationTokenRepository*

```
public interface VerificationTokenRepository extends JpaRepository<VerificationTok

    VerificationToken findByToken(String token);

    VerificationToken findByUser(User user);
}
```

Example 5.2. – The *IUserService* Interface

```
public interface IUserService {

    User registerNewUserAccount(UserDto accountDto) throws EmailExistsException;

    User getUser(String verificationToken);

    void saveRegisteredUser(User user);

    void addVerificationToken(User user, String token);

    VerificationToken getVerificationToken(String VerificationToken);

    void deleteUser(User user);
}
```

Example 5.3. The *UserService*

```
@Service
public class UserService implements IUserService {
    @Autowired
    private UserRepository repository;

    @Autowired
    private VerificationTokenRepository tokenRepository;

    @Transactional
    @Override
    public User registerNewUserAccount(UserDto accountDto) throws EmailExistsExcep
        if (emailExist(accountDto.getEmail())) {
            throw new EmailExistsException("There is an account with that email ad
              accountDto.getEmail());
        }
        User user = new User();
```

```
        user.setFirstName(accountDto.getFirstName());
        user.setLastName(accountDto.getLastName());
        user.setPassword(accountDto.getPassword());
        user.setEmail(accountDto.getEmail());
        user.setRole(new Role(Integer.valueOf(1), user));
        return repository.save(user);
    }

    private boolean emailExist(String email) {
        User user = repository.findByEmail(email);
        if (user != null) {
            return true;
        }
        return false;
    }

    @Override
    public User getUser(String verificationToken) {
        User user = tokenRepository.findByToken(verificationToken).getUser();
        return user;
    }

    @Override
    public VerificationToken getVerificationToken(String VerificationToken) {
        return tokenRepository.findByToken(VerificationToken);
    }

    @Transactional
    @Override
    public void saveRegisteredUser(User user) {
        repository.save(user);
    }

    @Transactional
    @Override
    public void deleteUser(User user) {
        repository.delete(user);
    }

    @Transactional
    @Override
    public void addVerificationToken(User user, String token) {
        VerificationToken myToken = new VerificationToken(token, user);
        tokenRepository.save(myToken);
    }
}
```

6. Conclusion

We have expanded our Spring registration process to include an email based account activation procedure. The account activation logic requires sending a verification token to the user via email, so that he can send it back to the controller to verify his identity. A *Spring event handler layer* takes care of the back-end work needed to send the confirmation email after the controller persists a registered.

http://twitter.com/share

Subscribe

Subscribe to our e-mail newsletter to receive updates.

http://inprogress.baeldung.com/?feed=rss2

(published) Handling Static Resources With Spring [http://inprogress.baeldung.com/?p=653]

Convert HTML to PDF using Apache FOP [http://inprogress.baeldung.com/?p=1430]

No comments yet.

Leave a Reply Click here to cancel reply. [/?p=1092#respond]

Logged in as odeskAuthor8 [http://inprogress.baeldung.com/wp-admin/profile.php]. Log out? [http://inprogress.baeldung.com/wp-login.php?action=logout]

Comment