# 1.安全控制Spring Security

## 1.Spring 对Spring Security的支持

### 1.1 什么是Spring Security

Spring Security是专门针对基于Spring的项目的安全框架,充分利用了依赖注入和AOP来实现安全的功能.

安全框架有两个重要概念,即认证(Authentication)和授权(Authorization).认证即确认用户可以访问当前系统,授权即确定用户在当前系统下所有的功能权限

### 1.2 Spring Security的配置

(1) DelegatingFilterProxy

Spring Security为我们提供了一个多个过滤器来实现所有安全功能,只需要注册一个特殊的DelegatingFilterProxy过滤器到WebApplicationInitializer即可.

```
public class WebInitializer implements WebApplicationInitializer {
   @Override
   public void onStartup(ServletContext servletContext) throws ServletException {
      AnnotationConfigWebApplicationContext context = new AnnotationConfigWebApplicationContext();
      context.register(MyMvcConfig.class,MySecutiryConfig.class);
      context.setServletContext(servletContext);

      ServletRegistration.Dynamic servlet = servletContext.addServlet("dispatcher", new DispatcherServlet(context));
      servlet.addMapping("/");
      servlet.setAsyncSupported(true);
      servlet.setLoadOnStartup(1);

      /*spring security拦截器代理*/
      DelegatingFilterProxy filterProxy = new DelegatingFilterProxy();
      //名字必须是: springSecurityFilterChain
      FilterRegistration.Dynamic filter = servletContext.addFilter("springSecurityFilterChain", filterProxy);
      filter.addMappingForUrlPatterns(null,false,"/*");
   }
}
```

(2)配置

Spring Security配置类:  ch9_1演示了使用数据库用户完整验证

```
@Configuration
@EnableWebSecurity
public class MySecutiryConfig extends WebSecurityConfigurerAdapter {
   @Override
   protected void configure(HttpSecurity http) throws Exception {
      //缺了这一行会报错
      //http.authorizeRequests().anyRequest().authenticated();
      http.authorizeRequests()
            .anyRequest().authenticated()
      .and().formLogin()
            .loginPage("/userLogin")
            .defaultSuccessUrl("/list")
            .failureUrl("/login?error")
            .permitAll()
            .and()
            .rememberMe()
            .tokenValiditySeconds(1209600)
            .and().logout().permitAll(); //cookie有效期 2周
      http.csrf().disable();//csrf攻击
   }

   //4
   @Override
   protected void configure(AuthenticationManagerBuilder auth) throws Exception {
      auth
            .inMemoryAuthentication().passwordEncoder(new MyPasswordEncoder())
            .withUser("txx").password("123").roles("USER");
   }

   //5忽略静态资源的拦截
   @Override
   public void configure(WebSecurity web) throws Exception {
      web.ignoring().antMatchers("/resources/static/**");
```

```
    }
}
```

### 1.3 用户认证

重写

```
@Override
protected void configure(AuthenticationManagerBuilder auth) throws Exception {
    auth.userDetailsService(customUserService()).passwordEncoder(passwordEncoder()); //3
}
```

(1) 内存中的用户

```
@Override
protected void configure(AuthenticationManagerBuilder auth) throws Exception {
    auth.inMemoryAuthentication().passwordEncoder(new MyPasswordEncoder())
            .withUser("txx").password("123").roles("USER");
}
```

(2) JDBC中的用户

JDBC中的用户直接指定dataSource即可

```
@Autowired
DataSource dataSource;
@Override
protected void configure(AuthenticationManagerBuilder auth) throws Exception {
    auth.jdbcAuthentication().dataSource(datasource);
}
```

Spring Security默认了数据库结构,如果需要自定义查询用户和权限的SQL语句:

```
auth.jdbcAuthentication().dataSource(datasource)
        .usersByUsernameQuery("select username,password,true from myusers where username=?")
        .authoritiesByUsernameQuery("select username,role form roles where username=?");
```

(3) 通用用户

上面的用户的获取方式仅限于内存或者JDBC,我们的数据访问形式各种各样

这时需要我们自定义实现UserDetailsService接口, 参见ch9_1

```
public class CustomUserService implements UserDetailsService { //1
  @Autowired
  SysUserRepository userRepository;

  @Override
  public UserDetails loadUserByUsername(String username) { //2
    SysUser user = userRepository.findByUsername(username);
    if(user == null){
      throw new UsernameNotFoundException("用户名不存在");
    }
    return user; //3
  }
}
```

### 1.4请求授权

通过重写:

```
protected void configure(HttpSecurity http) throws Exception {}
```

Spring Security使用以下匹配器来匹配请求路径:

1. antMathchers:使用ant风格的路径匹配
2. regexMatchers:使用正则表达式匹配路径

安全处理方法

| 方法 | 用途 |
| --- | --- |
| access(String) | Spring EL表达式结果为true时可访问 |
| anonymous() | 匿名可访问,只有匿名用户可以访问,认证用户访问不了 |
| denyAll() | 用户不能访问 |
| fullyAuthenticated() | 用户完全认证可访问(非remember me下自动登陆) |
| hasAnyAuthority(String...) | 如果用户有参数,则其中任一权限可访问 |
| hasAnyRole(String...) | 如果用户有参数,则其中任一角色可访问 |
| hasAuthority(String) | 如果用户有参数,则其权限可以访问 |
| hasIpAddress(String) | 如果用户来自参数中的IP则可访问 |

| | |
|---|---|
| hasRole(String) | 若用户有参数中的角色可访问 |
| permitAll() | 用户可任意访问 |
| remeberMe() | 允许通过remember-me登陆的用户访问 |
| authenticated() | 用户登陆后可访问 |

#### 1.5 定制登陆行为

```
http.formLogin() //1 通过formLogin方法定制登陆操作
   .loginPage("/login") //2 使用loginPage方法制定登陆页面的访问地址
   .defaultSuccessUrl("/index") //3 指定登陆成功后转向的页面
   .failureUrl("/login?error") //4 指定登陆失败后转向的页面
   .permitAll()
   .and()
   .rememberMe()//5 开启cookie存储用户信息
   .tokenValiditySeconds(1209600) // 2周免登陆
   .key("myKey") // 指定cookie中的私钥
   .logout() // 定制注销行为
   .logoutUrl("/custom-logout") //指定注销的url路径
   .logoutSuccessUrl("/logout-success") //指定注销成功后转向的页面
   .permitAll()
```

### 2.Spring Boot对Spring Security的支持

主要通过SecurityAutoConfiguration和SecurityProperties来完成配置,我们自动获得如下的自动配置:

1 ) 自动配置一个内存中的用户,账号为user,密码在程序启动时出现

2 ) 忽略/css/**,/js/**,/images/**和/**/favicon.ico等静态文件的拦截

3 ) 自动忽略配置的securityFilterChainRegistration的Bean

扩展配置时,只需要配置类继承WebSecurityConfigurerAdapter类即可,无需使用@EnableWebSecurity,但是需要使用@Configuration

# 2.批处理Spring Batch

## 2.1Spring Batch入门

#### 2.1.1 什么是Spring Batch

Spring Batch是用来处理大量数据操作的一个框架,主要用来读取大量数据,然后进行一定处理后输出成指定的形式.

#### 2.1.2 Spring Batch主要组成

SpringBatch主要由下面组成:

| 名称 | 用途 |
|---|---|
| JobRepository | 用来注册Job的容器 |
| JobLauncher | 用来启动Job的接口 |
| Job | 我们要实际执行的任务,包含一个或多个Step |
| Step | Step-步骤包含ItemReader,ItemProcessor和ItemWriter |
| ItemReader | 用来读取数据的接口 |
| ItemProcessor | 用来处理数据的接口 |
| ItemWriter | 用来输出数据的接口 |

以上Spring Batch的主要组成部分,只需要注册成Spring的Bean,即可,若想开启批处理的支持还需再配置类上使用@EnableBatchProcessing.

具体配置参考ch9_2_spring_batch

#### 2.1.3 Job监听

若需要监听我们的Job的执行情况,则定义一个类实现JobExecutionListener,并在定义Job的Bean上绑定该监听器.

#### 2.1.4 数据读取

Spring Batch为我们提供了大量的ItemReader实现,用来读取不同的数据来源.

#### 2.1.5 数据处理及校验

数据处理和校验都要通过ItemProcessor接口实现来完成.

(1) 数据处理

数据处理只需要实现ItemProcessor接口,重写其process方法.方法输入的参数是从ItemReader读取到的数据,返回的数据给ItemWriter.

(2) 数据校验

我们可以JSR-303(主要实现有hibernate-validator)的注解,来校验ItemReader读取到的数据是否满足要求.

可以让ItemProcessor实现ValidatingItemProcessor接口.

参考ch9_2_spring_batch

#### 2.1.6 数据输出

Spring Batch为我们提供了大量的ItemWriter的实现,用来输出到不同的目的地.

#### 2.1.7 计划任务

Spring Batch的任务是通过JobLauncher的run方法来执行的,因此我们只需在普通的计划任务方法中执行JobLauncher的run方法即可.

使用@EnableScheduling开启计划任务的支持

#### 2.1.8 参数后置绑定

在ItemReader和ItemWriter的Bean定义的时候,参数已经硬编码在Bean初始化中,读取的文件路径硬编码在Bean的定义中,不符合实际情况,这时候需要参数后置绑定.

要实现参数后置绑定,我们可以在JobParameters中绑定参数,在Bean定义的时候使用一个特殊的Bean生命周期注解@StepScope,然后通过@Value注入此参数.

# 3. 异步消息

异步消息主要目的是为了系统与系统之间的通信,所谓异步消息即消息发送者无需等待消息接收者的处理及返回,甚至无需关心消息是否发送成功.

异步消息中有两个很重要的概念,即消息代理(message broker)和目的地(destination).当消息发送者发送消息后,消息将由消息代理接管,消息代理保证消息传递到指定的目的地

异步消息主要有两种形式的目的地:队列(queue)和主题(topic).队列用于点对点式的消息通信;主题用于发布/订阅(publish/subscribe)的消息通信.

#### 1.点对点式

当消息发送者发送消息,消息代理获得消息后将消息放进一个队列里,当有消息接收者来接收消息的时候,消息将从队列取出来传递给接收者,这时候队列里就没有了这条消息

点对点式确保的是每一条消息只有唯一的发送者和接收者,但这并不能说明只有一个接收者可以从队列里接收消息,因为队列里有多个消息,点对点只保证每一条消息只有唯一的发送者和接收者

#### 2.发布/订阅式

发布/订阅是消息发送者发送消息到主题(topic),而多个消息接收者监听这个主题,

## 3.1 企业级消息代理

JMS(Java Message Service)即Java消息服务,是基于JVM消息代理的规范,而ActiveMQ,HornetQ是一个JMS消息代理的实现.

AMQP(Advanced Message Queuing Protocol)也是一个消息代理的规范,但它不仅兼容JMS,还支持跨语言和平台.AMQP的主要实现由RabbitMQ.

## 3.2 Spring的支持

Spring对JMS和AMQP的支持分别来自于Spring-jms和Spring-rabbit.

它们分别需要ConnectionFactory的实现来连接消息代理,并分别提供了JmsTemplate,RabbitTemplate来发送消息.

Spring为JMS,AMQP提供了@JmsListener,@RabbitListener注解在方法上监听消息代理发布的消息,我们需要分别通过@EnableJms,@EnableRabbit开启支持.

## 3.3 Spring Boot的支持

以ActiveMQ为例,Spring Boot为我们定义了ActiveMQConnectionFactory的Bean作为连接,并通过spring.activemq为前缀的属性来配置ActiveMQ的连接属性.

Spring Boot在JmsAutoConfiguration还为我们配置好了JmsTemplate,且为我们开启了注解式消息监听的支持,即自动开启了@EnableJms.

## 3.4 JMS实战

### 3.4.1 ActiveMQ

1) 消息定义

定义JMS发送的消息需实现MessageCreator接口,并重写createMessage方法.

```
public class Msg implements MessageCreator{
  @Override
  public Message createMessage(Session session) throws JMSException {
    return session.createTextMessage("测试消息");
```

```
  }
}
```

2）消息发送及目的地定义

```
@SpringBootApplication
public class Ch93ActivemqApplication implements CommandLineRunner {//1
  @Autowired
  JmsTemplate jmsTemplate; //2

  public static void main(String[] args) {
    SpringApplication.run(Ch93ActivemqApplication.class, args);
  }

   @Override
   public void run(String... args) throws Exception {
     jmsTemplate.send("my-destination", new Msg()); //3
   }
}
#1 Spring Boot为我们提供了CommandLineRunner接口,用于程序启动后执行的代码,
   通过重写run方法执行
#2 注入Spring Boot为我们配置好的JmsTemplate的Bean
#3 通过JmsTemplate的send方法向"my-destination"目的地发送Msg的消息,这里也等于
   在消息代理上定义了一个目的地叫my-destination
```

3）消息监听

```
@Component
public class Receiver {
  @JmsListener(destination = "my-destination") //1
  public void receiveMessage(String message) {
    System.out.println("接受到: <" + message + ">");
  }
}
#1 @JmsListener是Spring4.1为我们提供的一个新特性,用来简化JMS开发.
   我们只需在这个注解的属性destination指定要监听的目的,即可接收该目的地发送的消息
```

Spring5.0以上,需要使用jms2.0版本的jar包,Spring-jms没有自动导入JMS2.0,需要添加依赖,连个包:1.javax-jms.jar;2.javaxjms-api.jar

这里遇到了问题:使用docker 创建容器时警告:

WARNING: IPv4 forwarding is disabled. Networking will not work.

需要修改文件:

```
vim /usr/lib/sysctl.d/00-system.conf
重启network
systemctl restart network
删除错误activemq镜像容器
重新建立activemq镜像容器
```

## 3.5 AMQP实战

（1）定义信息及目的地定义

```
@SpringBootApplication
public class Ch93RabbitmqApplication implements CommandLineRunner {
  @Autowired
  RabbitTemplate rabbitTemplate; //1
  public static void main(String[] args) {
    SpringApplication.run(Ch93RabbitmqApplication.class, args);
  }
  @Bean //2
  public Queue wiselyQueue(){
    return new Queue("my-queue");
  }
  @Override
  public void run(String... args) throws Exception {
    rabbitTemplate.convertAndSend("my-queue", "来自RabbitMQ的问候"); //3
  }
}

#1 可注入Spring Boot为我们自动配置好的RabbitTemplate
#2 定义目的地队列,队列名称为"my-queue"
#3 通过RabbitTemplate的ConverterAndSend方法向队列发送消息
```

（2）消息监听

```
@Component
public class Receiver {
```

```
  @RabbitListener(queues = "my-queue")
   public void receiveMessage(String message) {
      System.out.println("Received <" + message + ">");
   }
}
```

# 通过@RabbitListener来监听RabbitMQ的目的地发送消息,
通过queues属性指定要监听的目的地

# 4. 系统集成Spring Integration

## 4.1 Spring Integration入门

Spring Integration主要由Message,Channel和Message EndPoint组成.

## 4.2 Message

Message是用来在不同部分之间传递的数据,Message由两部分组成:消息体(payload)与消息头(header).消息体可以是任何数据类型(xml,json,Java对象);消息头表示的元数据就是解释消息体的内容的.

## 4.3 Channel

在消息系统中,消息发送者发送消息到通道(Channel),消息接收者从通道(Channel)接收消息.

### 1.顶级接口

(1) MessageChannel

MessageChannel是Spring Integration消息通道的顶级接口,

当使用send方法发送消息时,返回值为true,则代表发送消息成功.MessageChannel有两大子接口,分别为PoolableChannel(可轮询)和SubScribleChannel(可订阅).我们所有的消息通道类都是实现者两个接口.

(2) PollableChannel

PollableChannel具备轮询获得消息的能力,

(3) SubscribableChannel

SubscribableChannel发送消息给订阅了MessageHanlder的订阅者

### 2.常用消息通道

(1) PublishSubscribeChannel

PublishSubscribeChannel允许广播消息给所有订阅者

(2) QueueChannel

QueueChannel允许消息接收者轮询获得信息,用一个队列(queue)接收消息,队列的容量大小可配置,

(3) PriorityChannel

PriorityChannel可按照优先级将数据存储到队列,它依据于消息的消息头priority属性

(4) RendezvousChannel

RendezvousChannel确保每一个接收者都接收到消息后再发送消息

(5) DirectChannel

DirectChannel是Spring Integration默认的消息通道,它允许将消息发送给每一个订阅者,然后阻碍发送直到消息被接收.

(6) ExecutorChannel

ExecutorChannel可绑定一个多线程的task executor

### 3.通道拦截器

Spring Integration给消息通道提供了通道拦截器,用来拦截发送和接收消息的操作

实现ChannelInterceptor接口

然后:

```
channel.addInterceptor(someInterceptor)
```

## 4.4 Message EndPoint

消息端点(Message EndPoint)是真正处理消息的(Message)组件,他还可以控制通道的路由,消息端点包含如下:

(1) Channel Adapter

通道适配器(Channel Adapter)是一种连接外部系统或传输协议的端点(EndPoint),可以分为入站(inbound)和出站(outbound)

通道适配器是单向的,入站通道适配器只支持接收消息,出站通道适配器只支持输出消息.

Spring Integration内置了多种适配器:

RabbitMQ,Feed,File,FTP/SFTP,Gemfire,HTTP,TCP/UDP,JDBC,JPA,JMS,Mail,MongoDB,Redis,RMI,Twitter,XMPP,WebService(SOAP,

等.

(2) Gateway

消息网关(Gateway)类似于Adapter,但是提供了双向的请求/返回集成方式,也分为入站(inbound)和出站(outbound).Spring Integration对相应的Adapter多都提供了Gateway.

(3) ServiceActivator

ServiceActivator可调用Spring的Bean来处理消息,并将处理后的结果输出到指定的消息通道.

(4) Router

路由(Router)可根据消息体类型(Payload Type Router),消息头的值(Header Value Router)以及定义好的接收表(Recipient List Router)作为条件,来决定消息传递到的通道.

(5) Filter

过滤器(Filter)类似于路由(Router),不同的是过滤器不决定消息路由到哪里,而是决定消息是否可以传递给消息通道

(6) Splitter

拆分器(Splitter)将消息拆分为几部分单独处理,拆分器处理的返回值是一个集合或者数组

(7) Aggregator

聚合器(Aggregator)与拆分器相反,它接收一个java.utile.List作为参数,将多个消息合并为一个消息.

(8) Enricher

当我们从外部获得消息后,需要增加额外的消息到已有的消息中,这是就需要使用消息增强器(Enricher).消息增强器主要由消息体增强器(Payload Enricher)和消息头增强器(Header Enricher)

(9) Transformer

转换器(Transformer)是对获得的消息进行一定处理的逻辑转换处理(如数据格式转换)

(10) Bridge

使用连接器(Bridge)可以简单地将两个消息通道连接起来.

## 4.5 Spring Integration Java DSL

Spring Integration提供了一个IntegrationFlow来定义系统集成流程,而通过IntegrationFlows和IntegrationFlowBuilder来实现使用Fluent API来定义流程,在Fluent API里,分别提供了下面方法来映射Spring integration地端点

```
transform() -> Transformer
filter() -> Filter
handle() -> ServiceActivator,Adapter,Gateway
split() -> Splitter
aggregate() -> Aggregator
route() -> Router
bridge() -> Bridge
```

简单的流程定义如下:

```
@Bean
public IntegrationFlow demoFlow(){
   return IntegrationFlows.from("input")
   .<String,Integer>transform(Integer::parseInt)
   .get(8)
}
```