# 常见Web安全漏洞

## XSS攻击

### 什么是XSS攻击手段

XSS攻击使用Javascript脚本注入进行攻击

例如在提交表单后，展示到另一个页面，可能会受到XSS脚本注入，读取本地cookie远程发送给黑客服务器端。

<script>alert('sss')</script>

<script>window.location.href='http://www.itmayiedu.com';</script>

对应html源代码: &lt;script&gt;alert('sss')&lt;/script&gt;

**最好使用火狐浏览器演示效果**

### 如何防御XSS攻击

将脚本特殊字符，转换成html源代码进行展示。

汉子编码http://www.mytju.com/classcode/tools/encode\_gb2312.asp

步骤：编写过滤器拦截所有getParameter参数，重写httpservletwrapp方法

将参数特殊字符转换成html源代码保存.

|  |
| --- |
| // 重写HttpServletRequestWrapper 防止XSS攻击  **public** **class** XssHttpServletRequestWrapper **extends** HttpServletRequestWrapper {  **private** HttpServletRequest request;  /\*\*  \* **@param** request  \*/  **public** XssHttpServletRequestWrapper(HttpServletRequest request) {  **super**(request);  **this**.request = request;  }  @Override  **public** String getParameter(String name) {  // 过滤getParameter参数 检查是否有特殊字符  String value = **super**.getParameter(name);  System.***out***.println("value:" + value);  **if** (!StringUtils.*isEmpty*(value)) {  // 将中文转换为字符编码格式，将特殊字符变为html源代码保存  value = StringEscapeUtils.*escapeHtml*(value);  System.***out***.println("newValue:" + value);  }  **return** value;  }  } |

SpringBoot启动加上@ServletComponentScan

|  |
| --- |
| @SpringBootApplication  @ServletComponentScan  **public** **class** App {  **public** **static** **void** main(String[] args) {  SpringApplication.*run*(App.**class**, args);  }  } |

## SQL注入攻击

### 什么是SQL注入

SQL注入：利用现有应用程序，将（恶意）的SQL命令注入到后台数据库执行一些恶意的

作。

造成SQL注入的原因是因为程序没有有效过滤用户的输入，使攻击者成功的向服务器提交恶意的SQL查询代码，程序在接收后错误的将攻击者的输入作为查询语句的一部分执行，导致原始的查询逻辑被改变，额外的执行了攻击者精心构造的恶意代码

### SQL注入防攻击手段

不要使用拼接SQL语句方式、最好使用预编译方式，在mybatis编写sql语句的时候，最好使用?传参数方式，不要使用#传参数，因为#传参数方式，可能会受到sql语句攻击。

演示案例:

http://127.0.0.1:8080/login?userName='liusi'&password='123'

http://127.0.0.1:8080/login?userName='liusi'&password='123' or 1=1

|  |
| --- |
| @RestController  **public** **class** LoginController {  @Autowired  **private** UserMapper userMapper;  @RequestMapping("/login")  **public** String login(UserEntity userEntity) {  System.***out***.println("账号密码信息:userEntity:" + userEntity.toString());  UserEntity login = userMapper.login(userEntity);  **return** login == **null** ? "登陆失败!" : "登陆成功!";  }  }  **public** **interface** UserMapper {  @Select(" SELECT \* FROM user\_info where userName=${userName} and password=${password}")  **public** UserEntity login(UserEntity userEntity);  } |

### MyBatis #与?区别

#{}: 解析为一个 JDBC 预编译语句（prepared statement）的参数标记符，一个 #{ } 被解析为一个参数占位符,可以防止SQL注入问题。

${}: 仅仅为一个纯碎的 string 替换，在动态 SQL 解析阶段将会进行变量替换。

## Http请求防盗链

### 什么是防盗链

比如A网站有一张图片，被B网站直接通过img标签属性引入，直接盗用A网站图片展示。

### 如何实现防盗链

判断http请求头Referer域中的记录来源的值，如果和当前访问的域名不一致的情况下，说明该图片可能被其他服务器盗用。

#### 使用过滤器判断请求头Referer记录请求来源

|  |
| --- |
| @WebFilter(filterName = "imgFilter", urlPatterns = "/imgs/\*")  **public** **class** ImgFilter **implements** Filter {  @Value("${domain.name}")  **private** String domainName;  **public** **void** init(FilterConfig filterConfig) **throws** ServletException {  }  **public** **void** doFilter(ServletRequest request, ServletResponse response, FilterChain chain)  **throws** IOException, ServletException {  HttpServletRequest req = (HttpServletRequest) request;  String referer = req.getHeader("Referer");  **if** (StringUtils.*isEmpty*(referer)) {  request.getRequestDispatcher("/imgs/error.png").forward(request, response);  **return**;  }  String domain = getDomain(referer);  **if** (!domain.equals(domainName)) {  request.getRequestDispatcher("/imgs/error.png").forward(request, response);  **return**;  }  chain.doFilter(request, response);  }  /\*\*  \* 获取url对应的域名  \*  \* **@param** url  \* **@return**  \*/  **public** String getDomain(String url) {  String result = "";  **int** j = 0, startIndex = 0, endIndex = 0;  **for** (**int** i = 0; i < url.length(); i++) {  **if** (url.charAt(i) == '/') {  j++;  **if** (j == 2)  startIndex = i;  **else** **if** (j == 3)  endIndex = i;  }  }  result = url.substring(startIndex + 1, endIndex);  **return** result;  }  **public** **void** destroy() {  }  } |

注意测试的时候,最好开启两个不同的浏览器测试，避免图片缓存的原因

## CSRF攻击

### CSRF攻击产生的原因

（Cross Site Request Forgery, 跨站域请求伪造）是一种网络的攻击方式，它在 2007 年曾被列为互联网 20 大安全隐患之一,也被称为“One Click Attack”或者Session Riding，通常缩写为CSRF或者XSRF，是一种对网站的恶意利用也就是人们所知道的钓鱼网站。尽管听起来像跨站脚本（XSS），但它与XSS非常不同，并且攻击方式几乎相左。XSS利用站点内的信任用户，而CSRF则通过伪装来自受信任用户的请求来利用受信任的网站。与XSS攻击相比，CSRF攻击往往不大流行（因此对其进行防范的资源也相当稀少）和难以防范，所以被认为比XSS更具危险性。

### API接口幂等性设计

#### API接口幂等性设计方案

##### MVCC方案

　多版本并发控制，该策略主要使用 update with condition（更新带条件来防止）来保证多次外部请求调用对系统的影响是一致的。在系统设计的过程中，合理的使用乐观锁，通过 version 或者 updateTime（timestamp）等其他条件，来做乐观锁的判断条件，这样保证更新操作即使在并发的情况下，也不会有太大的问题。例如

select \* from tablename where condition=#condition# // 取出要跟新的对象，带有版本 versoin

update tableName set name=#name#,version=version+1 where version=#version#

在更新的过程中利用 version 来防止，其他操作对对象的并发更新，导致更新丢失。为了避免失败，通常需要一定的重试机制。

##### 去重表

在插入数据的时候，插入去重表，利用数据库的唯一索引特性，保证唯一的逻辑。

##### 悲观锁

select for update，整个执行过程中锁定该订单对应的记录。注意：这种在 DB 读大于写的情况下尽量少用。

##### Token机制,防止页面重复提交

　业务要求：页面的数据只能被点击提交一次

　　发生原因：由于重复点击或者网络重发，或者 nginx 重发等情况会导致数据被重复提交

解决办法：

集群环境：采用 token 加 redis（redis 单线程的，处理需要排队）

单 JVM 环境：采用 token 加 redis 或 token 加 jvm 内存

处理流程：

数据提交前要向服务的申请 token，token 放到 redis 或 jvm 内存，token 有效时间

提交后后台校验 token，同时删除 token，生成新的 token 返回

token 特点:要申请，一次有效性，可以限流

#### 基于Token方式防止API接口幂等

客户端每次在调用接口的时候,需要在请求头中，传递令牌参数，每次令牌只能用一次。

一旦使用之后，就会被删除，这样可以有效防止重复提交。

步骤:

1.生成令牌接口

2. 接口中获取令牌验证

##### 生成令牌接口

|  |
| --- |
| **public** **class** TokenUtils {  **private** **static** Map<String, Object> *tokenMap* = **new** ConcurrentHashMap<String, Object>();  // 获取token  **public** **static** **synchronized** String getToken() {  // 1.生成令牌  String token = "token-" + System.*currentTimeMillis*();  // 2.存入tokenMap  *tokenMap*.put(token, token);  **return** token;  }  // 验证token,并且删除对应的token  **public** **static** Boolean exisToken(String token) {  // 1.从集合中获取token  Object result = *tokenMap*.get(token);  **if** (result == **null**) {  **return** **false**;  }  // 2.删除对应的token  *tokenMap*.remove(token);  **return** **true**;  }  } |

##### 接口中获取令牌验证

|  |
| --- |
| @RestController  **public** **class** OrderController {  @Autowired  **private** OrderMapper orderMapper;  // 获取Token  @RequestMapping("/getToken")  **public** String getToken() {  **return** TokenUtils.*getToken*();  }  // 验证Token  @RequestMapping(value = "/addOrder", produces = "application/json; charset=utf-8")  **public** String addOrder(@RequestBody OrderEntity orderEntity, HttpServletRequest request) {  String token = request.getHeader("token");  **if** (StringUtils.*isEmpty*(token)) {  **return** "参数错误!";  }  **if** (!TokenUtils.*exisToken*(token)) {  **return** "请勿重复提交!";  }  **int** result = orderMapper.addOrder(orderEntity);  **return** result > 0 ? "添加成功" : "添加失败" + "";  }  } |

### 防御CSRF攻击手段

使用图形验证码防止机器模拟接口请求攻击，在调用核心业务接口时，比如支付、下单、等接口，最好使用手机短信验证验证或者是人脸识别，防止其他用户使用Token伪造请求。

## 忘记密码漏洞

黑客使用抓包工具分析Http请求，在忘记密码找回时，需要发送一套短信验证码，如果验证码数字比较短的话，很容易使用暴力破解方式攻击破。

防御手段：

忘记密码验证码最好在6-8位。

一旦频繁调用接口验证时，应该使用图形验证码拦截，防止机器模拟。

使用黑名单和白名单机制，防御攻击。

## 上传文件漏洞

### 漏洞描述

上传漏洞这个顾名思义，就是攻击者通过上传木马文件，直接得到WEBSHELL，危害等级超级高，现在的入侵中上传漏洞也是常见的漏洞。

导致该漏洞的原因在于代码作者没有对访客提交的数据进行检验或者过滤不严，可以直接提交修改过的数据绕过扩展名的检验。

### 漏洞危害

1）可以得到WEBSHELL

2）上传木马文件，可以导致系统瘫痪

Tomcat虚拟地址:

F:\itmayiedujiangke2018-06-12\.metadata\.plugins\org.eclipse.wst.server.core\tmp0\wtpwebapps\tomcat\_web

### 环境搭建

#### Maven依赖

|  |
| --- |
| <dependencies>  <dependency>  <groupId>javax.servlet</groupId>  <artifactId>javax.servlet-api</artifactId>  <version>4.0.1</version>  </dependency>  <dependency>  <groupId>commons-fileupload</groupId>  <artifactId>commons-fileupload</artifactId>  <version>1.3.3</version>  </dependency>  <dependency>  <groupId>commons-io</groupId>  <artifactId>commons-io</artifactId>  <version>2.4</version>  </dependency>  </dependencies> |

#### Index.jsp

|  |
| --- |
| <%@ page language=*"java"* contentType=*"text/html; charset=UTF-8"*  pageEncoding=*"UTF-8"*%>  <!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN" "http://www.w3.org/TR/html4/loose.dtd">  <html>  <head>  <meta http-equiv=*"Content-Type"* content=*"text/html; charset=UTF-8"*>  <title>Insert title here</title>  </head>  <body>  <form action=*"/load/UploadServlet"* method=*"post"*  enctype=*"multipart/form-data"*>  <input type=*"file"* name=*"file"* /> <input type=*"submit"* value=*"submit"* />  </form>  </body>  </html> |

#### UploadServlet

|  |
| --- |
| **public** **class** UploadServlet**extends** HttpServlet {  /\*\*  \* 文件上传  \*/  **protected** **void** doPost(HttpServletRequest request, HttpServletResponse response) {  String root = request.getServletContext().getRealPath("/upload");  DiskFileItemFactory factory = **new** DiskFileItemFactory();  ServletFileUpload upload = **new** ServletFileUpload(factory);  **try** {  List<FileItem> list = upload.parseRequest(request);  **for** (FileItem it : list) {  // 如果是file文件类型  **if** (!it.isFormField()) {  it.write(**new** File(root + "/" + it.getName()));  response.getWriter().write("success");  }  }  } **catch** (Exception e) {  **try** {  response.getWriter().write("exception");  } **catch** (IOException e1) {  e1.printStackTrace();  }  e.printStackTrace();  }  }  } |

#### 脚本文件

#### 创建一个a.jsp

|  |
| --- |
| <%@page import="java.io.File"%>  <%@ page language="java" contentType="text/html; charset=utf-8" pageEncoding="utf-8"%>  <%  String root ="F:\\itmayiedu";  File file = new File(root);  file.delete();  %> |

### 修复方案

1)对文件格式限制，只允许某些格式上传

2)对文件格式进行校验，前端跟服务器都要进行校验（前端校验扩展名，服务器校验扩展名、Content\_Type等）

3)将上传目录防止到项目工程目录之外，当做静态资源文件路径，并且对文件的权限进行设定，禁止文件下的执行权限。

#### 判断文件流是否为图片格式

|  |
| --- |
| /\*\*  \* 文件上传  \*/  **protected** **void** doPost(HttpServletRequest request, HttpServletResponse response) {  String root = request.getServletContext().getRealPath("/upload");  DiskFileItemFactory factory = **new** DiskFileItemFactory();  ServletFileUpload upload = **new** ServletFileUpload(factory);  **try** {  List<FileItem> list = upload.parseRequest(request);  **for** (FileItem it : list) {  // 如果是file文件类型  **if** (!it.isFormField()) {  FileType fileType = *getFileType*(it.getInputStream());  **if** (fileType == **null**) {  // 非图片格式  response.getWriter().write("fail");  **return**;  }  String imgValue = fileType.getValue();  System.***out***.println("imgValue:" + imgValue);  // 是图片格式  it.write(**new** File(root + "/" + it.getName()));  response.getWriter().write("success");  }  }  } **catch** (Exception e) {  **try** {  response.getWriter().write("exception");  } **catch** (IOException e1) {  e1.printStackTrace();  }  e.printStackTrace();  }  }  // 判断文件是图片格式  **public** **static** FileType getFileType(InputStream is) **throws** IOException {  **byte**[] src = **new** **byte**[28];  is.read(src, 0, 28);  StringBuilder stringBuilder = **new** StringBuilder("");  **if** (src == **null** || src.length <= 0) {  **return** **null**;  }  **for** (**int** i = 0; i < src.length; i++) {  **int** v = src[i] & 0xFF;  String hv = Integer.*toHexString*(v).toUpperCase();  **if** (hv.length() < 2) {  stringBuilder.append(0);  }  stringBuilder.append(hv);  }  FileType[] fileTypes = FileType.*values*();  **for** (FileType fileType : fileTypes) {  **if** (stringBuilder.toString().startsWith(fileType.getValue())) {  **return** fileType;  }  }  **return** **null**;  } |

## 其他攻击和漏洞

直接异常信息，会给攻击者以提示。 可以使用mvc中的工具，把错误码异常等进行封装

HTML注释， 会暴露功能，方便攻击。 上线时去除注释

文件上传， 如果本身功能就是上传文件去执行，那么就有可能执行非常危险的命令。 解决方式是，设置文件白名单，限制文件类型，另外还可以重新命名文件，改名为不可执行的

路径遍历， 使用相对路径来遍历未开放的目录。 方式是将JS,CSS部署在独立的服务器，使用独立域名。 其他文件不使用静态URL访问，动态参数不包含文件路径信息。

## 网站安全漏洞扫描

# 互联网API接口幂等设计

## BaseRedisService封装Redis

|  |
| --- |
| @Component  **public** **class** BaseRedisService {  @Autowired  **private** StringRedisTemplate stringRedisTemplate;  **public** **void** setString(String key, Object data, Long timeout) {  **if** (data **instanceof** String) {  String value = (String) data;  stringRedisTemplate.opsForValue().set(key, value);  }  **if** (timeout != **null**) {  stringRedisTemplate.expire(key, timeout, TimeUnit.***SECONDS***);  }  }  **public** Object getString(String key) {  **return** stringRedisTemplate.opsForValue().get(key);  }  **public** **void** delKey(String key) {  stringRedisTemplate.delete(key);  }  } |

## RedisTokenUtils工具类

|  |
| --- |
| @Component  **public** **class** RedisTokenUtils {  **private** **long** timeout = 60 \* 60;  @Autowired  **private** BaseRedisService baseRedisService;  // 将token存入在redis  **public** String getToken() {  String token = "token" + System.*currentTimeMillis*();  baseRedisService.setString(token, token, timeout);  **return** token;  }  **public** **boolean** findToken(String tokenKey) {  String token = (String) baseRedisService.getString(tokenKey);  **if** (StringUtils.*isEmpty*(token)) {  **return** **false**;  }  // token 获取成功后 删除对应tokenMapstoken  baseRedisService.delKey(token);  **return** **true**;  }  } |

## 自定义Api幂等注解和切面

|  |
| --- |
| @Target(value = ElementType.***METHOD***)  @Retention(RetentionPolicy.***RUNTIME***)  **public** **@interface** ExtApiIdempotent {  String value();  } |

|  |
| --- |
| @Aspect  @Component  **public** **class** ExtApiAopIdempotent {  @Autowired  **private** RedisTokenUtils redisTokenUtils;  @Pointcut("execution(public \* com.itmayiedu.controller.\*.\*(..))")  **public** **void** rlAop() {  }  @Around("rlAop()")  **public** Object doBefore(ProceedingJoinPoint proceedingJoinPoint) **throws** Throwable {  MethodSignature signature = (MethodSignature) proceedingJoinPoint.getSignature();  ExtApiIdempotent extApiIdempotent = signature.getMethod().getDeclaredAnnotation(ExtApiIdempotent.**class**);  **if** (extApiIdempotent == **null**) {  // 直接执行程序  Object proceed = proceedingJoinPoint.proceed();  **return** proceed;  }  // 代码步骤：  // 1.获取令牌 存放在请求头中  HttpServletRequest request = getRequest();  String token = request.getHeader("token");  **if** (StringUtils.*isEmpty*(token)) {  response("参数错误!");  **return** **null**;  }  // 2.判断令牌是否在缓存中有对应的令牌  // 3.如何缓存没有该令牌的话，直接报错（请勿重复提交）  // 4.如何缓存有该令牌的话，直接执行该业务逻辑  // 5.执行完业务逻辑之后，直接删除该令牌。  **if** (!redisTokenUtils.findToken(token)) {  response("请勿重复提交!");  **return** **null**;  }  Object proceed = proceedingJoinPoint.proceed();  **return** proceed;  }  **public** HttpServletRequest getRequest() {  ServletRequestAttributes attributes = (ServletRequestAttributes) RequestContextHolder.*getRequestAttributes*();  HttpServletRequest request = attributes.getRequest();  **return** request;  }  **public** **void** response(String msg) **throws** IOException {  ServletRequestAttributes attributes = (ServletRequestAttributes) RequestContextHolder.*getRequestAttributes*();  HttpServletResponse response = attributes.getResponse();  response.setHeader("Content-type", "text/html;charset=UTF-8");  PrintWriter writer = response.getWriter();  **try** {  writer.println(msg);  } **catch** (Exception e) {  } **finally** {  writer.close();  }  }  } |

## 幂等注解使用

|  |
| --- |
| // 从redis中获取Token  @RequestMapping("/redisToken")  **public** String RedisToken() {  **return** redisTokenUtils.getToken();  }  // 验证Token  @RequestMapping(value = "/addOrderExtApiIdempotent", produces = "application/json; charset=utf-8")  @ExtApiIdempotent  **public** String addOrderExtApiIdempotent(@RequestBody OrderEntity orderEntity, HttpServletRequest request) {  **int** result = orderMapper.addOrder(orderEntity);  **return** result > 0 ? "添加成功" : "添加失败" + "";  } |

## 封装生成token注解

|  |
| --- |
| @Target(value = ElementType.***METHOD***)  @Retention(RetentionPolicy.***RUNTIME***)  **public** **@interface** ExtApiToken {  } |

## 改造ExtApiAopIdempotent

|  |
| --- |
| @Aspect  @Component  **public** **class** ExtApiAopIdempotent {  @Autowired  **private** RedisTokenUtils redisTokenUtils;  @Pointcut("execution(public \* com.itmayiedu.controller.\*.\*(..))")  **public** **void** rlAop() {  }  // 前置通知转发Token参数  @Before("rlAop()")  **public** **void** before(JoinPoint point) {  MethodSignature signature = (MethodSignature) point.getSignature();  ExtApiToken extApiToken = signature.getMethod().getDeclaredAnnotation(ExtApiToken.**class**);  **if** (extApiToken != **null**) {  extApiToken();  }  }  // 环绕通知验证参数  @Around("rlAop()")  **public** Object doAround(ProceedingJoinPoint proceedingJoinPoint) **throws** Throwable {  MethodSignature signature = (MethodSignature) proceedingJoinPoint.getSignature();  ExtApiIdempotent extApiIdempotent = signature.getMethod().getDeclaredAnnotation(ExtApiIdempotent.**class**);  **if** (extApiIdempotent != **null**) {  **return** extApiIdempotent(proceedingJoinPoint, signature);  }  // 放行  Object proceed = proceedingJoinPoint.proceed();  **return** proceed;  }  // 验证Token  **public** Object extApiIdempotent(ProceedingJoinPoint proceedingJoinPoint, MethodSignature signature)  **throws** Throwable {  ExtApiIdempotent extApiIdempotent = signature.getMethod().getDeclaredAnnotation(ExtApiIdempotent.**class**);  **if** (extApiIdempotent == **null**) {  // 直接执行程序  Object proceed = proceedingJoinPoint.proceed();  **return** proceed;  }  // 代码步骤：  // 1.获取令牌 存放在请求头中  HttpServletRequest request = getRequest();  String valueType = extApiIdempotent.value();  **if** (StringUtils.*isEmpty*(valueType)) {  response("参数错误!");  **return** **null**;  }  String token = **null**;  **if** (valueType.equals(ConstantUtils.***EXTAPIHEAD***)) {  token = request.getHeader("token");  } **else** {  token = request.getParameter("token");  }  **if** (StringUtils.*isEmpty*(token)) {  response("参数错误!");  **return** **null**;  }  **if** (!redisTokenUtils.findToken(token)) {  response("请勿重复提交!");  **return** **null**;  }  Object proceed = proceedingJoinPoint.proceed();  **return** proceed;  }  **public** **void** extApiToken() {  String token = redisTokenUtils.getToken();  getRequest().setAttribute("token", token);  }  **public** HttpServletRequest getRequest() {  ServletRequestAttributes attributes = (ServletRequestAttributes) RequestContextHolder.*getRequestAttributes*();  HttpServletRequest request = attributes.getRequest();  **return** request;  }  **public** **void** response(String msg) **throws** IOException {  ServletRequestAttributes attributes = (ServletRequestAttributes) RequestContextHolder.*getRequestAttributes*();  HttpServletResponse response = attributes.getResponse();  response.setHeader("Content-type", "text/html;charset=UTF-8");  PrintWriter writer = response.getWriter();  **try** {  writer.println(msg);  } **catch** (Exception e) {  } **finally** {  writer.close();  }  }  } |

## API接口保证幂等性

|  |
| --- |
| @RestController  **public** **class** OrderController {  @Autowired  **private** OrderMapper orderMapper;  @Autowired  **private** RedisTokenUtils redisTokenUtils;  // 从redis中获取Token  @RequestMapping("/redisToken")  **public** String RedisToken() {  **return** redisTokenUtils.getToken();  }  // 验证Token  @RequestMapping(value = "/addOrderExtApiIdempotent", produces = "application/json; charset=utf-8")  @ExtApiIdempotent(value = ConstantUtils.***EXTAPIHEAD***)  **public** String addOrderExtApiIdempotent(@RequestBody OrderEntity orderEntity, HttpServletRequest request) {  **int** result = orderMapper.addOrder(orderEntity);  **return** result > 0 ? "添加成功" : "添加失败" + "";  }  } |

## 页面防止重复提交

|  |
| --- |
| @Controller  **public** **class** OrderPageController {  @Autowired  **private** OrderMapper orderMapper;  @RequestMapping("/indexPage")  @ExtApiToken  **public** String indexPage(HttpServletRequest req) {  **return** "indexPage";  }  @RequestMapping("/addOrderPage")  @ExtApiIdempotent(value = ConstantUtils.***EXTAPIFROM***)  **public** String addOrder(OrderEntity orderEntity) {  **int** addOrder = orderMapper.addOrder(orderEntity);  **return** addOrder > 0 ? "success" : "fail";  }  } |

# API安全接口安全设计

## 互联网开放平台设计

1.需求:现在A公司与B公司进行合作，B公司需要调用A公司开放的外网接口获取数据，

如何保证外网开放接口的安全性。

2.常用解决办法：

2.1 使用加签名方式，防止篡改数据

2.2 使用Https加密传输

2.3 搭建OAuth2.0认证授权

2.4 使用令牌方式

2.5 搭建网关实现黑名单和白名单

## Api接口访问演示地址

## 使用令牌方式搭建搭建API开放平台

原理：为每个合作机构创建对应的appid、app\_secret，生成对应的access\_token（有效期2小时），在调用外网开放接口的时候，必须传递有效的access\_token。

### 数据库表设计

|  |
| --- |
| CREATE TABLE `m\_app` (  `id` int(11) NOT NULL AUTO\_INCREMENT,  `app\_name` varchar(255) DEFAULT NULL,  `app\_id` varchar(255) DEFAULT NULL,  `app\_secret` varchar(255) DEFAULT NULL,  `is\_flag` varchar(255) DEFAULT NULL,  `access\_token` varchar(255) DEFAULT NULL,  PRIMARY KEY (`id`)  ) ENGINE=InnoDB AUTO\_INCREMENT=2 DEFAULT CHARSET=utf8; |

App\_Name 表示机构名称

App\_ID 应用id

App\_Secret 应用密钥 （可更改）

Is\_flag 是否可用 （是否对某个机构开放）

access\_token 上一次access\_token

### 获取AccessToken

|  |
| --- |
| // 创建获取getAccessToken  @RestController  @RequestMapping(value = "/auth")  **public** **class** AuthController **extends** BaseApiService {  @Autowired  **private** BaseRedisService baseRedisService;  **private** **long** timeToken = 60 \* 60 \* 2;  @Autowired  **private** AppMapper appMapper;  // 使用appId+appSecret 生成AccessToke  @RequestMapping("/getAccessToken")  **public** ResponseBase getAccessToken(AppEntity appEntity) {  AppEntity appResult = appMapper.findApp(appEntity);  **if** (appResult == **null**) {  **return** setResultError("没有对应机构的认证信息");  }  **int** isFlag = appResult.getIsFlag();  **if** (isFlag == 1) {  **return** setResultError("您现在没有权限生成对应的AccessToken");  }  // ### 获取新的accessToken 之前删除之前老的accessToken  // 从redis中删除之前的accessToken  String accessToken = appResult.getAccessToken();  baseRedisService.delKey(accessToken);  // 生成的新的accessToken  String newAccessToken = newAccessToken(appResult.getAppId());  JSONObject jsonObject = **new** JSONObject();  jsonObject.put("accessToken", newAccessToken);  **return** setResultSuccessData(jsonObject);  }  **private** String newAccessToken(String appId) {  // 使用appid+appsecret 生成对应的AccessToken 保存两个小时  String accessToken = TokenUtils.*getAccessToken*();  // 保证在同一个事物redis 事物中  // 生成最新的token key为accessToken value 为 appid  baseRedisService.setString(accessToken, appId, timeToken);  // 表中保存当前accessToken  appMapper.updateAccessToken(accessToken, appId);  **return** accessToken;  }  } |

### 编写拦截器拦截请求,验证accessToken

|  |
| --- |
| //验证AccessToken 是否正确  @Component  **public** **class** AccessTokenInterceptor **extends** BaseApiService **implements** HandlerInterceptor {  @Autowired  **private** BaseRedisService baseRedisService;  /\*\*  \* 进入controller层之前拦截请求  \*  \* **@param** httpServletRequest  \* **@param** httpServletResponse  \* **@param** o  \* **@return**  \* **@throws** Exception  \*/  **public** **boolean** preHandle(HttpServletRequest httpServletRequest, HttpServletResponse httpServletResponse, Object o)  **throws** Exception {  System.***out***.println("---------------------开始进入请求地址拦截----------------------------");  String accessToken = httpServletRequest.getParameter("accessToken");  // 判断accessToken是否空  **if** (StringUtils.*isEmpty*(accessToken)) {  // 参数Token accessToken  resultError(" this is parameter accessToken null ", httpServletResponse);  **return** **false**;  }  String appId = (String) baseRedisService.getString(accessToken);  **if** (StringUtils.*isEmpty*(appId)) {  // accessToken 已经失效!  resultError(" this is accessToken Invalid ", httpServletResponse);  **return** **false**;  }  // 正常执行业务逻辑...  **return** **true**;  }  **public** **void** postHandle(HttpServletRequest httpServletRequest, HttpServletResponse httpServletResponse, Object o,  ModelAndView modelAndView) **throws** Exception {  System.***out***.println("--------------处理请求完成后视图渲染之前的处理操作---------------");  }  **public** **void** afterCompletion(HttpServletRequest httpServletRequest, HttpServletResponse httpServletResponse,  Object o, Exception e) **throws** Exception {  System.***out***.println("---------------视图渲染之后的操作-------------------------0");  }  // 返回错误提示  **public** **void** resultError(String errorMsg, HttpServletResponse httpServletResponse) **throws** IOException {  PrintWriter printWriter = httpServletResponse.getWriter();  printWriter.write(**new** JSONObject().*toJSONString*(setResultError(errorMsg)));  }  } |

## URL转码

### 什么是URL转码

不管是以何种方式传递url时，如果要传递的url中包含特殊字符，如想要传递一个+，但是这个+会被url会被编码成空格，想要传递&，被url处理成分隔符。

尤其是当传递的url是经过Base64加密或者RSA加密后的，存在特殊字符时，这里的特殊字符一旦被url处理，就不是原先你加密的结果了。

url特殊符号及对应的编码:

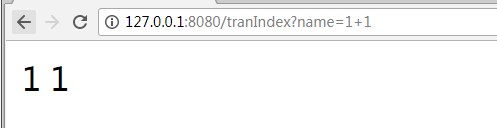
| **符号** | **url中的含义** | **编码** |
| --- | --- | --- |
| + | URL 中+号表示空格 | %2B |
| 空格 | URL中的空格可以用+号或者编码 | %20 |
| / | 分隔目录和子目录 | %2F |
| ? | 分隔实际的URL和参数 | %3F |
| % | 指定特殊字符 | %25 |
| # | 表示书签 | %23 |
| & | URL中指定的参数间的分隔符 | %26 |
| = | URL中指定参数的值 | %3D |

### URLEncode和URLDecode

接受参数案例tranIndex

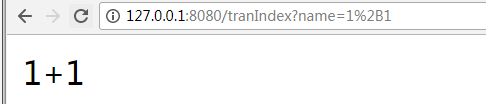
|  |
| --- |
| @RestController  **public** **class** TranController {  // 接受客户端参数  @RequestMapping("/tranIndex")  **public** String tranIndex(String name) {  System.***out***.println("name:" + name);  **return** name;  }  } |

客户端访问结果



传入+参数变为了空格。

解决办法：将+变为%2B



### Java代码处理转码

URLEncoder.encode和decode

|  |
| --- |
| String encode = URLEncoder.*encode*("1+1", "UTF-8");  String decode = URLDecoder.*decode*(encode, "UTF-8");  System.***out***.println("encode:" + encode + ",decode:" + decode); |

Http接口参数编码处理

|  |
| --- |
| String url = "http://127.0.0.1:8080/tranIndex?";  // 参数转码  String strParam = "name=" + URLEncoder.*encode*("1+1", "utf-8");  String newUrl = url + strParam;  String result = HttpClientUtils.*httpGet*(newUrl);  System.***out***.println("result:" + result); |

## 信息加密与密钥管理

### 单向散列加密

散列是信息的提炼，通常其长度要比信息小得多，且为一个固定长度。加密性强的散列一定是不可逆的，这就意味着通过散列结果，无法推出任何部分的原始信息。任何输入信息的变化，哪怕仅一位，都将导致散列结果的明显变化，这称之为雪崩效应。散列还应该是防冲突的，即找不出具有相同散列结果的两条信息。具有这些特性的散列结果就可以用于验证信息是否被修改。  
单向散列函数一般用于产生消息摘要，密钥加密等，常见的有：  
1、MD5（Message Digest Algorithm 5）：是RSA数据安全公司开发的一种单向散列算法，非可逆，相同的明文产生相同的密文。  
2、SHA（Secure Hash Algorithm）：可以对任意长度的数据运算生成一个160位的数值；  
SHA-1与MD5的比较  
因为二者均由MD4导出，SHA-1和MD5彼此很相似。相应的，他们的强度和其他特性也是相似，但还有以下几点不同：  
1、对强行供给的安全性：最显著和最重要的区别是SHA-1摘要比MD5摘要长32 位。使用强行技术，产生任何一个报文使其摘要等于给定报摘要的难度对MD5是2128数量级的操作，而对SHA-1则是2160数量级的操作。这样，SHA-1对强行攻击有更大的强度。  
2、对密码分析的安全性：由于MD5的设计，易受密码分析的攻击，SHA-1显得不易受这样的攻击。  
3、速度：在相同的硬件上，SHA-1的运行速度比MD5慢。

1、特征：雪崩效应、定长输出和不可逆。  
2、作用是：确保数据的完整性。  
3、加密算法：md5（标准密钥长度128位）、sha1（标准密钥长度160位）、md4、CRC-32  
4、加密工具：md5sum、sha1sum、openssl dgst。  
5、计算某个文件的hash值，例如：md5sum/shalsum FileName,openssl dgst –md5/-sha

#### MD5加密

##### 在线MD5解密与加密

<http://www.cmd5.com/>

##### Java操作MD5加密

##### MD5加盐实现方式

一般使用的加盐：

md5(Password+UserName)，即将用户名和密码字符串相加再MD5，这样的MD5摘要基本上不可反查。

但有时候用户名可能会发生变化，发生变化后密码即不可用了（验证密码实际上就是再次计算摘要的过程）。

----------

因此我们做了一个非常简单的加盐算法，每次保存密码到数据库时，都生成一个随机16位数字，将这16位数字和密码相加再求MD5摘要，然后在摘要中再将这16位数字按规则掺入形成一个48位的字符串。

在验证密码时再从48位字符串中按规则提取16位数字，和用户输入的密码相加再MD5。按照这种方法形成的结果肯定是不可直接反查的，且同一个密码每次保存时形成的摘要也都是不同的。

代码如下：

|  |
| --- |
| /\*\*  \* MD5加盐加密  \*/  **public** **class** PasswordUtil {  /\*\*  \* 生成含有随机盐的密码  \*/  **public** **static** String generate(String password) {  Random r = **new** Random();  StringBuilder sb = **new** StringBuilder(16);  sb.append(r.nextInt(99999999)).append(r.nextInt(99999999));  **int** len = sb.length();  **if** (len < 16) {  **for** (**int** i = 0; i < 16 - len; i++) {  sb.append("0");  }  }  String salt = sb.toString();  password = *md5Hex*(password + salt);  **char**[] cs = **new** **char**[48];  **for** (**int** i = 0; i < 48; i += 3) {  cs[i] = password.charAt(i / 3 \* 2);  **char** c = salt.charAt(i / 3);  cs[i + 1] = c;  cs[i + 2] = password.charAt(i / 3 \* 2 + 1);  }  **return** **new** String(cs);  }  /\*\*  \* 校验密码是否正确  \*/  **public** **static** **boolean** verify(String password, String md5) {  **char**[] cs1 = **new** **char**[32];  **char**[] cs2 = **new** **char**[16];  **for** (**int** i = 0; i < 48; i += 3) {  cs1[i / 3 \* 2] = md5.charAt(i);  cs1[i / 3 \* 2 + 1] = md5.charAt(i + 2);  cs2[i / 3] = md5.charAt(i + 1);  }  String salt = **new** String(cs2);  **return** *md5Hex*(password + salt).equals(**new** String(cs1));  }  /\*\*  \* 获取十六进制字符串形式的MD5摘要  \*/  **public** **static** String md5Hex(String src) {  **try** {  MessageDigest md5 = MessageDigest.*getInstance*("MD5");  **byte**[] bs = md5.digest(src.getBytes());  **return** **new** String(**new** Hex().encode(bs));  } **catch** (Exception e) {  **return** **null**;  }  }  **public** **static** **void** main(String[] args) {  // 加密+加盐  String password1 = *generate*("admin");  System.***out***.println("结果：" + password1 + " 长度：" + password1.length());  // 解码  System.***out***.println(*verify*("admin", password1));  // 加密+加盐  String password2 = *generate*("admin");  System.***out***.println("结果：" + password2 + " 长度：" + password2.length());  // 解码  System.***out***.println(*verify*("admin", password2));  }  } |

### 信息加密技术

#### 单向散列加密

#### 对称加密

#### 非对称加密

#### 安全密钥管理

#### 如何保证HTTP接口请求的安全呢？

# 电子商务风控系统

# 黑名单与白名单系统

# 基于多种方式实现防御DDOS攻击