Java面试题-Web相关

一.你怎么理解 RESTful?

2000 年，Roy Thomas Fielding 博士在他那篇著名的博士论文《Architectural Styles and the Design of Network-based Software Architectures》中提出了几种软件应用的架构风格，REST 作为其中的一种架构风格在这篇论文的第5章中进行了概括性的介绍。

REST 是“REpresentational State Transfer”的缩写，可以翻译成“表现状态转换”，但是在绝大多数场合中我们只说 REST 或者 RESTful。Fielding 在论文中将 REST 定位为“分布式超媒体应用（Distributed Hypermedia System）”的架构风格，它在文中提到一个名为“HATEOAS（Hypermedia as the engine of application state）”的概念。

我们利用一个面向最终用户的 Web 应用来对这个概念进行简单阐述：这里所谓的应用状态（Application State）表示 Web 应用的客户端的状态，简单起见可以理解为会话状态。资源在浏览器中以超媒体的形式呈现，通过点击超媒体中的链接可以获取其它相关的资源或者对当前资源进行相应的处理，获取的资源或者针对资源处理的响应同样以超媒体的形式再次呈现在浏览器上。由此可见，超媒体成为了驱动客户端会话状态的转换的引擎。

借助于超媒体这种特殊的资源呈现方式，应用状态的转换体现为浏览器中呈现资源的转换。如果将超媒体进一步抽象成一般意义上的资源呈现（Representation）方式，那么应用状态变成了可被呈现的状态（REpresentational State）。应用状态之间的转换就成了可被呈现的状态装换（REpresentational State Transfer），这就是 REST。

总结

REST 是一种很笼统的概念，它代表一种架构风格。

二.如何理解 RESTful API 的幂等性?

1.什么是幂等性?

HTTP 幂等方法，是指无论调用多少次都不会有不同结果的 HTTP 方法。不管你调用一次，还是调用一百次，一千次，结果都是相同的。

```
GET     /tickets        # 获取ticket列表
GET     /tickets/12     # 查看某个具体的ticket
POST    /tickets        # 新建一个ticket
PUT     /tickets/12     # 更新ticket 12
PATCH   /tickets/12     # 更新ticket 12
DELETE  /tickets/12     # 删除ticekt 12
```

2.HTTP GET 方法

HTTP GET 方法，用于获取资源，不管调用多少次接口，结果都不会改变，所以是幂等的。

```
GET     /tickets        # 获取ticket列表
GET     /tickets/12     # 查看某个具体的ticket
```

只是查询数据，不会影响到资源的变化，因此我们认为它幂等。

值得注意，幂等性指的是作用于结果而非资源本身。怎么理解呢？例如，这个 HTTP GET 方法可能会每次得到不同的返回内容，但并不影响资源。

可能你会问有这种情况么？当然有咯。例如，我们有一个接口获取当前时间，我们就应该设计成

```
GET     /service_time # 获取服务器当前时间
```

它本身不会对资源本身产生影响，因此满足幂等性。

3.HTTP POST 方法

HTTP POST 方法是一个非幂等方法，因为调用多次，都将产生新的资源。

```
POST    /tickets        # 新建一个ticket
```

因为它会对资源本身产生影响，每次调用都会有新的资源产生，因此不满足幂等性。

4.HTTP PUT 方法

HTTP PUT 方法是不是幂等的呢？我们来看下

```
PUT     /tickets/12    # 更新ticket 12
```

因为它直接把实体部分的数据替换到服务器的资源，我们多次调用它，只会产生一次影响，但是有相同结果的 HTTP 方法，所以满足幂等性。

5.HTTP PATCH 方法

HTTP PATCH 方法是非幂等的。HTTP POST 方法和 HTTP PUT 方法可能比较好理解，但是 HTTP PATCH 方法只是更新部分资源，怎么是非幂等的呢？

因为，PATCH 提供的实体则需要根据程序或其它协议的定义，解析后在服务器上执行，以此来修改服务器上的资源。换句话说，PATCH 请求是会执行某个程序的，如果重复提交，程序可能执行多次，对服务器上的资源就可能造成额外的影响，这就可以解释它为什么是非幂等的了。

可能你还不能理解这点。我们举个例子

```
PATCH   /tickets/12    # 更新ticket 12
```

此时，我们服务端对方法的处理是，当调用一次方法，更新部分字段，将这条 ticket 记录的操作记录加一，这次，每次调用的资源是不是变了呢，所以它是有可能是非幂等的操作。

6.HTTP DELETE 方法

HTTP DELETE 方法用于删除资源，会将资源删除。

```
DELETE  /tickets/12    # 删除ticekt 12
```

调用一次和多次对资源产生影响是相同的，所以也满足幂等性。

三.如何设计符合幂等性的高质量 RESTful API?

1.HTTP GET vs HTTP POST

也许，你会想起一个面试题。**HTTP 请求的 GET 与 POST 方式有什么区别？** 你可能会回答到：GET 方式通过 URL 提交数据，数据在 URL 中可以看到；POST 方式，数据放置在 HTML HEADER 内提交。但是，我们现在从 RESTful 的资源角度来看待问题，HTTP GET 方法是幂等的，所以它适合作为查询操作，HTTP POST 方法是非幂等的，所以用来表示新增操作。

但是，也有例外，我们有的时候可能需要把查询方法改造成 HTTP POST 方法。比如，超长（1k）的 GET URL 使用 POST 方法来替代，因为 GET 受到 URL 长度的限制。虽然，它不符合幂等性，但是它是一种折中的方案。

2.HTTP POST vs HTTP PUT

对于 HTTP POST 方法和 HTTP PUT 方法，我们一般的理解是 POST 表示创建资源，PUT 表示更新资源。当然，这个是正确的理解。

但是，实际上，两个方法都用于创建资源，更为本质的差别是在幂等性。HTTP POST 方法是非幂等，所以用来表示创建资源，HTTP PUT 方法是幂等的，因此表示更新资源更加贴切。

3.HTTP PUT vs HTTP PATCH

此时，你看会有另外一个问题。HTTP PUT 方法和 HTTP PATCH 方法，都是用来表述更新资源，它们之间有什么区别呢？我们一般的理解是 PUT 表示更新全部资源，PATCH 表示更新部分资源。首先，这个是我们遵守的第一准则。根据上面的描述，PATCH 方法是非幂等的，因此我们在设计我们服务端的 RESTful API 的时候，也需要考虑。如果，我们想要明确的告诉调用者我们的资源是幂等的，我的设计更倾向于使用 HTTP PUT 方法。

四.前后端分离是如何做的?

在前后端分离架构中，后端只需要负责按照约定的数据格式向前端提供可调用的 API 服务即可。前后端之间通过 HTTP 请求进行交互，前端获取到数据后，进行页面的组装和渲染，最终返回给浏览器。

五.说说如何设计一个良好的 API?

1.版本号

在 RESTful API 中，API 接口应该尽量兼容之前的版本。但是，在实际业务开发场景中，可能随着业务需求的不断迭代，现有的 API 接口无法支持旧版本的适配，此时如果强制升级服务端的 API 接口将导致客户端旧有功能出现故障。实际上，Web 端是部署在服务器，因此它可以很容易为了适配服务端的新的 API 接口进行版本升级，然而像 Android 端、IOS 端、PC 端等其他客户端是运行在用户的机器上，因此当前产品很难做到适配新的服务端的 API 接口，从而出现功能故障，这种情况下，用户必须升级产品到最新的版本才能正常使用。

为了解决这个版本不兼容问题，在设计 RESTful API 的一种实用的做法是使用版本号。一般情况下，我们会在 url 中保留版本号，并同时兼容多个版本。

```
【GET】  /v1/users/{user_id}  // 版本 v1 的查询用户列表的 API 接口
【GET】  /v2/users/{user_id}  // 版本 v2 的查询用户列表的 API 接口
```

现在，我们可以不改变版本 v1 的查询用户列表的 API 接口的情况下，新增版本 v2 的查询用户列表的 API 接口以满足新的业务需求，此时，客户端的产品的新功能将请求新的服务端的 API 接口地址。虽然服务端会同时兼容多个版本，但是同时维护太多版本对于服务端而言是个不小的负担，因为服务端要维护多套代码。这种情况下，常见的做法不是维护所有的兼容版本，而是只维护最新的几个兼容版本，例如维护最新的三个兼容版本。在一段时间后，当绝大多数用户升级到较新的版本后，废弃一些使用量较少的服务端的老版本API 接口版本，并要求使用产品的非常旧的版本的用户强制升级。

注意的是，“不改变版本 v1 的查询用户列表的 API 接口”主要指的是对于客户端的调用者而言它看起来是没有改变。而实际上，如果业务变化太大，服务端的开发人员需要对旧版本的 API 接口使用适配器模式将请求适配到新的API 接口上。

2.资源路径

RESTful API 的设计以资源为核心，每一个 URI 代表一种资源。因此，URI 不能包含动词，只能是名词。注意的是，形容词也是可以使用的，但是尽量少用。一般来说，不论资源是单个还是多个，API 的名词要以复数进行命名。此外，命名名词的时候，要使用小写、数字及下划线来区分多个单词。这样的设计是为了与 json 对象及属性的命名方案保持一致。例如，一个查询系统标签的接口可以进行如下设计。

```
【GET】  /v1/tags/{tag_id}
```

同时，资源的路径应该从根到子依次如下

```
/{resources}/{resource_id}/{sub_resources}/{sub_resource_id}/{sub_resource_property}
```

我们来看一个“添加用户的角色”的设计，其中“用户”是主资源，“角色”是子资源。

```
【POST】  /v1/users/{user_id}/roles/{role_id} // 添加用户的角色
```

有的时候，当一个资源变化难以使用标准的 RESTful API 来命名，可以考虑使用一些特殊的 actions 命名。

```
/{resources}/{resource_id}/actions/{action}
```

举个例子，“密码修改”这个接口的命名很难完全使用名词来构建路径，此时可以引入 action 命名。

```
【PUT】  /v1/users/{user_id}/password/actions/modify // 密码修改
```

3.请求方式

可以通过 GET、 POST、 PUT、 PATCH、 DELETE 等方式对服务端的资源进行操作。其中：

- GET：用于查询资源
- POST：用于创建资源
- PUT：用于更新服务端的资源的全部信息
- PATCH：用于更新服务端的资源的部分信息
- DELETE：用于删除服务端的资源。

这里，使用“用户”的案例进行回顾通过 GET、 POST、 PUT、 PATCH、 DELETE 等方式对服务端的资源进行操作。

```
【GET】          /users                  # 查询用户信息列表
【GET】          /users/1001             # 查看某个用户信息
```

```
【POST】          /users                  # 新建用户信息
【PUT】           /users/1001             # 更新用户信息(全部字段)
【PATCH】         /users/1001             # 更新用户信息(部分字段)
【DELETE】        /users/1001             # 删除用户信息
```

4.查询参数

RESTful API 接口应该提供参数，过滤返回结果。其中，offset 指定返回记录的开始位置。一般情况下，它会结合 limit 来做分页的查询，这里 limit 指定返回记录的数量。

```
【GET】  /{version}/{resources}/{resource_id}?offset=0&limit=20
```

同时，orderby 可以用来排序，但仅支持单个字符的排序，如果存在多个字段排序，需要业务中扩展其他参数进行支持。

```
【GET】  /{version}/{resources}/{resource_id}?orderby={field} [asc|de
sc]
```

为了更好地选择是否支持查询总数，我们可以使用 count 字段，count 表示返回数据是否包含总条数，它的默认值为 false。

```
【GET】  /{version}/{resources}/{resource_id}?count=[true|false]
```

上面介绍的 offset、 limit、 orderby 是一些公共参数。此外，业务场景中还存在许多个性化的参数。我们来看一个例子。

```
【GET】  /v1/categorys/{category_id}/apps/{app_id}?enable=[1|0]&os_ty
pe={field}&device_ids={field,field,…}
```

注意的是，不要过度设计，只返回用户需要的查询参数。此外，需要考虑是否对查询参数创建数据库索引以提高查询性能。

5.状态码

使用适合的状态码很重要，而不应该全部都返回状态码 200，或者随便乱使用。这里，列举在实际开发过程中常用的一些状态码，以供参考。

状态码	描述
200	请求成功
201	创建成功
400	错误的请求
401	未验证
403	被拒绝
404	无法找到
409	资源冲突
500	服务器内部错误

6.异常响应

当 RESTful API 接口出现非 2xx 的 HTTP 错误码响应时，采用全局的异常结构响应信息。

```
HTTP/1.1 400 Bad Request
Content-Type: application/json
{
    "code": "INVALID_ARGUMENT",
    "message": "{error message}",
    "cause": "{cause message}",
    "request_id": "01234567-89ab-cdef-0123-456789abcdef",
    "host_id": "{server identity}",
    "server_time": "2014-01-01T12:00:00Z"
}
```

7.请求参数

在设计服务端的 RESTful API 的时候，我们还需要对请求参数进行限制说明。例如一个支持批量查询的接口，我们要考虑最大支持查询的数量。

```
【GET】      /v1/users/batch?user_ids=1001,1002      // 批量查询用户信息
参数说明
- user_ids: 用户ID串，最多允许 20 个。
```

此外，在设计新增或修改接口时，我们还需要在文档中明确告诉调用者哪些参数是必填项，哪些是选填项，以及它们的边界值的限制。

```
【POST】      /v1/users                                  // 创建用户信息
请求内容
{
    "username": "lusifer",                  // 必填，用户名称，max 10
    "realname": "鲁斯菲尔",                 // 必填，用户名称，max 10
    "password": "123456",                 // 必填，用户密码，max 32
    "email": "topsale@vip.qq.com",        // 选填，电子邮箱，max 32
    "weixin": "Lusifer",                // 选填，微信账号，max 32
    "sex": 1                              // 必填，用户性别[1-男 2-女 99-
未知]
}
```

8.响应参数

针对不同操作，服务端向用户返回的结果应该符合以下规范。

```
【GET】      /{version}/{resources}/{resource_id}      // 返回单个资源对
象
【GET】      /{version}/{resources}                    // 返回资源对象的
列表
【POST】     /{version}/{resources}                    // 返回新生成的资
源对象
【PUT】      /{version}/{resources}/{resource_id}      // 返回完整的资源
对象
【PATCH】    /{version}/{resources}/{resource_id}      // 返回完整的资源
对象
【DELETE】   /{version}/{resources}/{resource_id}      // 状态码 200,
返回完整的资源对象。
                                                       // 状态码 204,
返回一个空文档
```

如果是单条数据，则返回一个对象的 JSON 字符串。

```
HTTP/1.1 200 OK
{
    "id" : "01234567-89ab-cdef-0123-456789abcdef",
    "name" : "example",
    "created_time": 1496676420000,
    "updated_time": 1496676420000,
    ...
```

```
}
```

如果是列表数据，则返回一个封装的结构体。

```
HTTP/1.1 200 OK
{
    "count":100,
    "items":[
        {
            "id" : "01234567-89ab-cdef-0123-456789abcdef",
            "name" : "example",
            "created_time": 1496676420000,
            "updated_time": 1496676420000,
            ...
        },
        ...
    ]
}
```

9.一个完整的案例

最后，我们使用一个完整的案例将前面介绍的知识整合起来。这里，使用“获取用户列表”的案例。

```
【GET】      /v1/users?[&keyword=xxx][&enable=1][&offset=0][&limit=20]
 获取用户列表
功能说明：获取用户列表
请求方式：GET
参数说明
- keyword: 模糊查找的关键字。[选填]
- enable: 启用状态[1-启用 2-禁用]。[选填]
- offset: 获取位置偏移，从 0 开始。[选填]
- limit: 每次获取返回的条数，缺省为 20 条，最大不超过 100。 [选填]
响应内容
HTTP/1.1 200 OK
{
    "count":100,
    "items":[
        {
            "id" : "01234567-89ab-cdef-0123-456789abcdef",
            "name" : "example",
            "created_time": 1496676420000,
            "updated_time": 1496676420000,
            ...
```

```
        },
        ...
    ]
}
失败响应
HTTP/1.1 403 UC/AUTH_DENIED
Content-Type: application/json
{
    "code": "INVALID_ARGUMENT",
    "message": "{error message}",
    "cause": "{cause message}",
    "request_id": "01234567-89ab-cdef-0123-456789abcdef",
    "host_id": "{server identity}",
    "server_time": "2014-01-01T12:00:00Z"
}
错误代码
- 403 UC/AUTH_DENIED    授权受限
```

六.如何解决跨域?

1.什么是跨域问题?

跨域，指的是浏览器不能执行其他网站的脚本。它是由浏览器的同源策略造成的，是浏览器对 JavaScript 施加的安全限制。

2.什么是同源?

所谓同源是指，域名，协议，端口均相同

- `http://www.funtl.com --> http://admin.funtl.com 跨域`
- `http://www.funtl.com --> http://www.funtl.com 非跨域`
- `http://www.funtl.com --> http://www.funtl.com:8080 跨域`
- `http://www.funtl.com --> https://www.funtl.com 跨域`

3.使用 CORS（跨资源共享）解决跨域问题

CORS 是一个 W3C 标准，全称是"跨域资源共享"（Cross-origin resource sharing）。它允许浏览器向跨源服务器，发出 XMLHttpRequest 请求，从而克服了 AJAX 只能同源使用的限制。

CORS 需要浏览器和服务器同时支持。目前，所有浏览器都支持该功能，IE 浏览

器不能低于 IE10

整个 CORS 通信过程，都是浏览器自动完成，不需要用户参与。对于开发者来说，CORS 通信与同源的 AJAX 通信没有差别，代码完全一样。浏览器一旦发现 AJAX 请求跨源，就会自动添加一些附加的头信息，有时还会多出一次附加的请求，但用户不会有感觉

因此，实现 CORS 通信的关键是服务器。只要服务器实现了 CORS 接口，就可以跨源通信

4.CORS 与 JSONP 的比较

CORS 与 JSONP 的使用目的相同，但是比 JSONP 更强大。

JSONP 只支持 GET 请求，CORS 支持所有类型的 HTTP 请求。JSONP 的优势在于支持老式浏览器，以及可以向不支持 CORS 的网站请求数据。

七.安全要素与 STRIDE 威胁

STRIDE 威胁

STRIDE 威胁，代表六种安全威胁：

- 身份假冒（Spoofing）
- 篡改（Tampering）
- 抵赖（Repudiation）
- 信息泄露（Information Disclosure）
- 拒绝服务（Denial of Service）
- 特权提升（Elevation of Privilege）

1.身份假冒（Spoofing）

身份假冒，即伪装成某对象或某人。例如，我们通过伪造别人的 ID 进行操作

2.篡改（Tampering）

篡改，即未经授权修改数据或者代码。例如，我通过网络抓包或者某种途径修改某个请求包，而服务端没有进行进一步的防范措施，使得我篡改的请求包提交成功。

3.抵赖（Repudiation）

抵赖，即拒绝执行他人无法证实也无法反对的行为而产生抵赖。例如，我攻击了某个产品，他们并不知道是我做的，没有证据证明是我做的，我就可以进行抵赖，换句话说，我可以死不承认。

4.信息泄露（Information Disclosure）

信息泄露，即将信息暴露给未授权用户。例如，我通过某种途径获取未经加密的敏感信息，例如用户密码。

5.拒绝服务（Denial of Service）

拒绝服务，即拒绝或降低有效用户的服务级别。例如，我通过拒绝服务攻击，使得其他正常用户无法使用产品的相关服务功能。

6.特权提升（Elevation of Privilege）

特权提升，即通过非授权方式获得更高权限。例如，我试图用管理员的权限进行业务操作。

.安全要素

为了防范上面的 STRIDE 威胁，我们需要采用一些防范措施：

威胁	安全要素	消减技术
身份假冒	认证	Kerberos、SSL/TLS、证书、认证码等
篡改	完整性	访问控制列表、SSL/TLS、认证码等
抵赖	非抵赖/审计/记录	安全审计和日志记录、数字签名、可信第三方
信息泄露	保密	加密、访问控制列表
拒绝服务	可用性	访问控制列表、过滤、配额、授权
特权提升	授权	访问控制列表、角色控制、授权

八.什么是 TCP 粘包/拆包?

- 要发送的数据大于 TCP 发送缓冲区剩余空间大小，将会发生拆包。
- 待发送数据大于 MSS（最大报文长度），TCP 在传输前将进行拆包。
- 要发送的数据小于 TCP 发送缓冲区的大小，TCP 将多次写入缓冲区的数据一次发送出去，将会发生粘包。
- 接收数据端的应用层没有及时读取接收缓冲区中的数据，将发生粘包。

九.TCP 粘包/拆包的解决办法

- 发送端给每个数据包添加包首部，首部中应该至少包含数据包的长度，这样接收端在接收到数据后，通过读取包首部的长度字段，便知道每一个数据包的实际长度了。
- 发送端将每个数据包封装为固定长度（不够的可以通过补 0 填充），这样接收端每次从接收缓冲区中读取固定长度的数据就自然而然的把每个数据包拆分开来。
- 可以在数据包之间设置边界，如添加特殊符号，这样，接收端通过这个边界就可以将不同的数据包拆分开。

十.防范常见的 Web 攻击

1.SQL 注入攻击

SQL 注入攻击，这个是最常聊到的话题，使用过 Java 的开发人员，第一个反应就是一定要使用预编译的 PrepareStatement

什么是 SQL 注入攻击

攻击者在 HTTP 请求中注入恶意的 SQL 代码，服务器使用参数构建数据库 SQL 命令时，恶意 SQL 被一起构造，并在数据库中执行。

用户登录，输入用户名 Lusifer，密码 `'or '1' = '1'` ，如果此时使用参数构造的方式，就会出现

```
select * from user where name = 'Lusifer' and password = '' or '1'='1
'
```

不管用户名和密码是什么内容，使查询出来的用户列表不为空。

现在还会存在 SQL 注入攻击么

这个问题在使用了预编译的 PrepareStatement 后，安全性得到了很大的提高，但是真实情况下，很多同学并不重视，还是会留下漏洞的。举个例子，看看，大家的代码中对 sql 中 in 操作，使用了预编译，还是仍然还是通过字符串拼接呢?

如何防范 SQL 注入攻击

使用预编译的 PrepareStatement 是必须的，但是一般我们会从两个方面同时入手:

1. Web 端

- 有效性检验。
- 限制字符串输入的长度。

2. 服务端

- 不用拼接 SQL 字符串。
- 使用预编译的 PrepareStatement。
- 有效性检验。(为什么服务端还要做有效性检验? 第一准则，外部都是不可信的，防止攻击者绕过 Web 端请求)
- 过滤 SQL 需要的参数中的特殊字符。比如单引号、双引号。

2.XSS 攻击

什么是 XSS 攻击

跨站点脚本攻击，指攻击者通过篡改网页，嵌入恶意脚本程序，在用户浏览网页时，控制用户浏览器进行恶意操作的一种攻击方式。

假设页面上有一个表单

```
<input  type="text" name="name" value="Lusifer"/>
```

如果，用户输入的不是一个正常的字符串，而是

```
"/><script>alert("haha")</script><!-
```

此时，页面变成下面的内容，在输入框 input 的后面带上了一段脚本代码。

```
<input type="text" name="name" value="Lusifer"/><script>alert("haha")
</script><!-"/>
```

这端脚本程序只是弹出一个消息框，并不会造成什么危害，攻击的威力取决于用户输入了什么样的脚本，只要稍微修改，便可使攻击极具攻击性。

如何防范 XSS 攻击

1. 前端，服务端，同时需要字符串输入的长度限制。
2. 前端，服务端，同时需要对HTML转义处理。将其中的 `<` , `>` 等特殊字符进行转义编码。

3.CSRF 攻击

什么是 CSRF 攻击

跨站点请求伪造，指攻击者通过跨站请求，以合法的用户的身份进行非法操作。可以这么理解 CSRF 攻击：攻击者盗用你的身份，以你的名义向第三方网站发送恶意请求。CRSF 能做的事情包括利用你的身份发邮件，发短信，进行交易转账，甚至盗取账号信息。

如何防范 CSRF 攻击

1. 安全框架，例如 Spring Security。
2. token 机制。在 HTTP 请求中进行 token 验证，如果请求中没有 token 或者 token 内容不正确，则认为 CSRF 攻击而拒绝该请求。
3. 验证码。通常情况下，验证码能够很好的遏制 CSRF 攻击，但是很多情况下，出于用户体验考虑，验证码只能作为一种辅助手段，而不是最主要的解决方案。
4. referer 识别。在 HTTP Header 中有一个字段 Referer，它记录了 HTTP 请求的来源地址。如果 Referer 是其他网站，就有可能是 CSRF 攻击，则拒绝该请求。但是，服务器并非都能取到 Referer。很多用户出于隐私保护的考虑，限制了 Referer 的发送。在某些情况下，浏览器也不会发送 Referer，例如 HTTPS 跳转到 HTTP。

4.文件上传漏洞

什么是文件上传漏洞

文件上传漏洞，指的是用户上传一个可执行的脚本文件，并通过此脚本文件获得了执行服务端命令的能力。

许多第三方框架、服务，都曾经被爆出文件上传漏洞，比如很早之前的Struts2，以及富文本编辑器等等，可能被一旦被攻击者上传恶意代码，有可能服务端就被人黑了。

如何防范文件上传漏洞

1. 文件上传的目录设置为不可执行。
2. 判断文件类型。在判断文件类型的时候，可以结合使用 `MIME Type`，后缀检查等方式。因为对于上传文件，不能简单地通过后缀名称来判断文件的类型，因为攻击者可以将可执行文件的后缀名称改为图片或其他后缀类型，诱导用户执行。
3. 对上传的文件类型进行白名单校验，只允许上传可靠类型。
4. 上传的文件需要进行重新命名，使攻击者无法猜想上传文件的访问路径，将极大地增加攻击成本，同时向 `shell`，`php`，`rar`，`ara` 这种文件，因为重命名而无法成功实施攻击。
5. 限制上传文件的大小。
6. 单独设置文件服务器的域名。

5.访问控制

一般来说，“基于 URL 的访问控制”是最常见的。

垂直权限管理

访问控制实际上是建立用户与权限之间的对应关系，即“基于角色的访问控制”，RBAC。不同角色的权限有高低之分。高权限角色访问低权限角色的资源往往是被允许的，而低权限角色访问高权限的资源往往被禁止的。在配置权限时，应当使用“最小权限原则”，并使用“默认拒绝”的策略，只对有需要的主体单独配置”允许”的策略，这在很多时候能够避免发生“越权访问”。

例如，Spring Security， Apache Shiro 都可以建立垂直权限管理。

水平权限管理

水平权限问题在同一个角色上，系统只验证了访问数据的角色，没有对角色内的用户做细分，由于水平权限管理是系统缺乏一个数据级的访问控制所造成的，因此水平权限管理又可以称之为“基于数据的访问控制”。

举个理解，比如我们之前的一个助手产品，客户端用户删除评论功能，如果没有做水平权限管理，即设置只有本人才可以删除自己的评论，那么用户通过修改评论id就可以删除别人的评论这个就存在危险的越权操作。

这个层面，基本需要我们业务层面去处理，但是这个也是最为经常遗落的安全点。

十一.服务端通信安全攻防

服务端接口通信过程中，一般是明文传输的，没有经过任何安全处理。那么这个时候就很容易在传输过程中被中间者窃听、篡改、冒充等风险。因此，对于敏感信息，以及重要文件就需要进行加密策略，保证通信的安全性。

Base64 加密传输

Base64 是网络上最常见的用于传输 8Bit 字节代码的编码方式之一，但是它其实并不是一种用于安全领域的加密解密算法。

但是，Base64 编码的数据并不会被人用肉眼所直观的理解，所以也有人使用 Base64 来进行加密解密，这里所说的加密与解密实际是指编码和解码的过程。

这种，加密传输的安全性是非常低的，Base64 加密非常容易被人识别并解码。

DES 对称加密

DES 也是一种非常常用的加密方案，我们会将敏感的信息在通信过程中通过DES进行加密传输，然后在客户端和服务端直接进行解码。

此时，作为读者的你，可能会有个疑问，那如何保管密钥呢？其实，想想，答案就复出水面了，因为客户端和服务端都需要进行解码，所以两者都要存一份密钥。其实，还有一种方案是通过服务端下发，但是下发的时候通信的安全性也是没有很好的保障。

所以，DES 对称加密也是存在一定的安全隐患：密钥可能会泄漏。这边，举个真

实的案例，某个 APP 的资源不错，同事想抓包分析下其服务端通信的信息结构，但是发现它既然全部采用了 DES 加密方案，本来想放弃了，但是我们又回头想想客户端肯定需要密钥对接口的加密的内容做解码才能正常展现，那么密钥肯定在 APP 包中，因此我们又对 APP 进行了反编译，结果成功的获取到了密钥，对服务端通信的加密信息进行了解码。

AES 对称加密

AES 和 DES 类似，相较于 DES 算法而言，AES 算法有着更高的速度和资源使用效率，安全级别也较之更高。一般情况下，用于文件的加密。我们之前做个不准确测试，AES 和 DES 分别对一个大文件加密，AES 的速度大概是 DES 的 5 倍。（因为基于工具和环境问题，这个数据不是很准确哟）。

仍然存在一个相同的问题：密钥可能会泄漏。因此，保管好密钥很关键。

升级到 HTTPS

HTTPS 的价值在于：

- 内容加密，第三方无法窃听。
- 身份认证，一旦被篡改，通信双方会立刻发现。
- 数据完整性。防止内容冒充或者篡改。

这个方案，没法保护敏感数据，如果需要对敏感数据进行加密，还是需要考虑加密方案。

URL 签名

基于 OAuth2 协议，进行 URL 签名。这个方案，有很多话题可以分享，后面另开一篇来详细讲解。

值得注意的是，URL 签名只能垂直权限管理，但没法保护敏感数据，如果需要对敏感数据进行保护，还是需要考虑加密方案。

双向 RSA 加密

RSA 双向认证，就是用对方的公钥加密是为了保密，这个只有对方用私钥能解密。用自己的私钥加密是为了防抵赖，能用我的公钥解开，说明这是我发来的。

例如，支付宝的支付接口就是非常典型的 RSA 双向认证的安全方案。此外，我们之前的教育资源、敏感验证码出于安全性考虑都借鉴了这个方案。

十二.授权与认证

认证与授权是计算机安全方面的两个基础概念。认证解决你是谁的问题 (who)，授权解决你能干什么的问题 (what)。计算机系统解决你是谁的问题，是依靠识别与人绑定的某种凭证来做判断的，比如通过判定预设的用户名和密码是否匹配，或者读取人持有的 IC 卡、RFID、NFC 等信息来识别人的身份，当然随着科技的发展计算机的外设已经可以识别生物特征比如指纹识别、虹膜、人脸等特征。当然，计算机系统的用户不仅仅是人，也可能是其他的机器或系统，所以有必要对各种认证的手段做一个抽象，基于声明 (claims-based) 的认证就是这样一种抽象。

1.一个登机的例子

解释基于声明的认证这个概念之前先介绍一个大家比较熟悉的机场登机的场景。假定你准备从上海到北京出差并且已经通过携程、艺龙或者去哪儿买了一张从上海到北京的机票。那么接下来的流程是这样：

1. 换登机牌。到值机柜台出示必要证件，工作人员核实你的身份以及购票记录给你一张登机牌。登机牌除了显示姓名、性别、航班号等基础信息外还包括登机口、登机时间等专有信息。登机牌还包含有条码可以被机场的专有扫描设备识别。
2. 安全检查，用登机牌以及必要证件通过安全检查进入候机大厅。
3. 候机及登机。根据登机牌指示登记时间和登机口等候并登机。

2.基本概念

身份 (Identity)

标识一个用户或者一个实体。在上面的例子中，身份是坐飞机的人。

声明 (Claim)

Identity 的一系列属性比如姓名、email、角色等。在上面的例子中，声明指的是登机牌中展示的信息包括示姓名、性别、航班号等基础信息以及登机口、登机时间等专有信息。

安全令牌 (Security Token)

将 Claim 的信息通过数字签名等技术手段得到的一个载体，由发行机构（机场）颁发。在上面的例子中，安全令牌就是登机牌，里面包含条形码等不可伪造的信息可以被机场的专有设备识别。通常在计算机系统中令牌可以仅仅是一个加密的

信息，通过令牌可以向发行机构请求申明信息。

发行机构 (Issuing Authroty)

签发安全令牌的受信机构，可以是 Web 应用或 Web 服务。在上面的例子中发行机构是机场，它颁发登机牌。

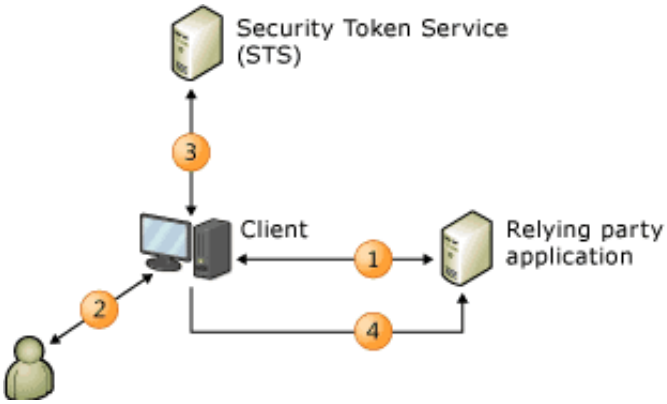
安全令牌服务 (Security Token Service)

提供构建安全令牌、签名、验证等服务。在上面的例子中，机场提供的值机台、安检口、登机口就是这种类型的服务。

信任方应用 (Repling Party Application)

信任发行机构，使用其颁发的安全令牌获得 Claim 的应用。在上面的例子中，这个信任方应用就是某航空公司提供的指定班次的飞行服务。

基于申明的认证模型



这幅图说明一个 Web 站点（信任方应用，relying party application, RP）使用基于申明的进行认证，被用户通过浏览器访问的过程。

1. 未授权用户访问 Web 站点的页面会被重定向到身份提供者的页面。
2. 身份提供者要求用户提供凭证，比如用户名和密码。
3. 身份提供者颁发安全令牌返回到浏览器。
4. 浏览器重定向到刚开始要访问的那个页面，Web 站点根据安全令牌决定是否允许访问。