架构安全性

- **认证**
  - **定义：**系统如何正确分辨出操作用户的真实身份。
  - **认证方式**
    - **通信信道上的认证：**建立通信连接之前，要先证明你是谁。在网络传输场景中的典型是基于 SSL/TLS 传输安全层的认证。
    - **通信协议上的认证：**请求获取资源之前，要先证明你是谁。在互联网场景中的典型是基于 HTTP 协议的认证。
    - **通信内容上的认证：**使用提供服务之前，要先证明你是谁。在万维网场景中的典型是基于 Web 内容的认证。
  - **HTTP认证**
    - **前提**
      - **认证方案**：是指生成用户身份凭证的某种方法，源于 HTTP 协议的认证框架
      - HTTP通用认证框架要求所有支持 HTTP 协议的服务器，**在未授权的用户意图访问服务端保护区域的资源时，应返回 401 Unauthorized，同时在响应报文头里附带以下两个分别代表网页认证和代理认证的 Header 之一**，告知客户端应该采取何种方式产生能代表访问者身份的凭证信息，**验证失败返回403 Forbidden 错误。**
    - **服务端响应Header**
      - **WWW-Authenticate: <认证方案> realm=<保护区域的描述信息>**
        Proxy-Authenticate: <认证方案> realm=<保护区域的描述信息>
    - **客户端请求Header**
      - **Authorization: <认证方案> <凭证内容>**
        Proxy-Authorization: <认证方案> <凭证内容>
    - **常见认证方案**
      - **HTTP Basic：**以演示为目的的认证方案，也应用于一些不要求安全性的场合。
      - **Digest：**HTTP 摘要认证，可视为 Basic 认证的改良版本，针对 Base64 明文发送的风险，把用户名和密码加盐后再通过 MD5/SHA 等哈希算法取摘要发送出去。
      - **Bearer：**基于 OAuth 2（同时涉及认证与授权的协议） 规范来完成认证。
      - **HOBA：**是一种基于自签名证书的认证方案。一类是采用 CA层次结构的模型，由 CA 中心签发证书；另一种是以 IETF 的 Token Binding 协议为基础的 OBC自签名证书模型。
      - **AWS4-HMAC-SHA256：**亚马逊 AWS 基于 HMAC-SHA256 哈希算法的认证。
      - **NTLM / Negotiate：**这是微软公司NTLM用到的两种认证方式。
      - **Windows Live ID：**微软开发并提供的“统一登入”认证。
      - **Twitter Basic：**Twitter网站所改良的 HTTP 基础认证。
  - **Web认证**
    - **背景：**以 HTTP 协议为基础的认证框架也只能面向传输协议设计，依靠内容来实现的认证方式肯定希望是由系统本身的功能去完成的。
    - **WebAuthn**
      - **背景：**万维网联盟W3C批准了由FIDO（安全、开放、防钓鱼、无密码认证标准的联盟）领导起草的世界首份 Web 内容认证的标准“WebAuthn”。直接采用生物识别或者实体密钥来作为身份凭证，从根本上消灭了用户输入错误产生的校验需求和防止机器人模拟产生的验证码需求等问题，甚至可以省掉表单界面。
      - **注册流程**
        - 1、服务端先暂存用户提交的用户信息，生成一个随机字符串Challenge和用户的 UserID，返回给客户端
        - 2、客户端的 WebAuthn API 接收到 Challenge 和 UserID，把这些信息发送给验证器，可理解为用户设备上 TouchID、FaceID、实体密钥等认证设备统一接口。
        - 3、验证器提示用户进行验证，结果是生成一个密钥对（公钥和私钥），由验证器存储私钥、用户信息以及当前的域名。然后使用私钥对 Challenge 进行签名，并将签名结果、UserID 和公钥一起返回客户端。
        - 4、服务器核验信息，检查 UserID 与之前发送的是否一致，并用公钥解密后得到的结果与之前发送的 Challenge 相比较，一致即表明注册通过，由服务端存储该 UserID 对应的公钥。
      - **登录流程**
        - 1、服务器返回随机字符串 Challenge、用户 UserID
        - 2、浏览器将 Challenge 和 UserID 转发给验证器
        - 3、验证器提示用户进行认证操作，通过后以存储的私钥加密 Challenge，然后返回给浏览器
        - 4、服务端接收到浏览器转发来的被私钥加密的 Challenge，以此前注册时存储的公钥进行解密，如果解密成功则宣告登录成功
      - **优势**
        - 采用非对称加密的公钥、私钥替代传统的密码，私钥是保密的，只有验证器需要知道它，没有人为泄漏的可能；除了得知私钥外，没有其他途径能够生成可被公钥验证为有效的签名，可以通过公钥是否能够解密来判断最终用户的身份是否合法。
        - 解决了传统密码在网络传输上的风险
        - 为登录过程带来极大的便捷性，而且彻底避免了用户在一个网站上泄漏密码，所有使用相同密码的网站都受到攻击的问题。
  - **JAVA安全框架**

- **主流框架**
  - **Apache Shiro**
  - **Spring Security**
- **功能**
  - **认证功能：**以 HTTP 协议中定义的各种认证、表单等认证方式确认用户身份。
  - **安全上下文：**用户获得认证之后，应用可以得知用户的基本资料、权限、角色等。
  - **授权功能：**判断并控制认证后的用户对什么资源拥有哪些操作许可
  - **密码的存储与验证：**密码是烫手的山芋，存储、传输还是验证都应谨慎处理

授权

- **定义：**系统如何控制一个用户该看到哪些数据、能操作哪些功能。
- **包含问题**
  - **确保授权的过程可靠：**对于单一系统来说，授权的过程是比较容易做到可控的。而在涉及多方的系统中，需要既让第三方系统能够访问到所需的资源，又能保证其不泄露用户的敏感数据。常用的多方授权协议主要有 OAuth2 和 SAML 2.0。
  - **确保授权的结果可控：**授权的结果用于对程序功能或者资源的访问控制，成理论体系的权限控制模型有自主访问控制DAC、强制访问控制MAC、基于属性的访问控制ABAC，还有最为常用的基于角色的访问控制RBAC。
- **RBAC**
  - **背景：**访问控制模型，实质上都是在解决“谁拥有什么权限去操作哪些资源”。为了避免对每一个用户设定权限，RBAC 将权限从用户身上剥离，改为绑定到“角色”上，将权限控制变为解决“角色拥有操作哪些资源的许可”。
  - **优势**
    - 不仅简化配置操作，还满足了“最小特权原则”。角色拥有许可的数量是根据完成该角色工作职责所需的最小权限来赋予的。
    - RBAC 允许对不同角色之间定义关联与约束，进一步强化它的抽象描述能力。如不同的角色之间可以有继承性，不同角色之间也可以具有互斥性。
  - **建立访问控制模型的基本目的是为了管理垂直权限和水平权限**。垂直权限即功能权限，水平权限即数据权限管理。数据权限是很难抽象与通用的，基本只能由信息系统自主来来完成。
- **OAuth2**
  - **直接将密码告知第三方应用出现的问题**
    - **密码泄漏：**如果第三方应用被黑客攻破，将导致密码也同时被泄漏。
    - **访问范围：**第三方应用将有能力读取、修改、删除、更新源服务端上的所有资源。
    - **授权回收：**只有修改密码才能回收授予给第三方应用的权力，授权的应用可能有许多，修改了密码意味着所有别的第三方的应用程序的授权会全部失效。
  - **解决问题方法：**以令牌Token代替用户密码作为授权的凭证。令牌被泄漏，也不会导致密码的泄漏；令牌上可以设定访问资源的范围以及时效性；每个应用都持有独立的令牌，哪个失效都不会波及其他。
  - **授权码模式**
    - **授权过程**
      - 1、第三方应用将资源所有者（用户）导向授权服务器的授权页面，并向授权服务器提供 ClientID 及用户同意授权后的回调 URI，这是一次客户端页面转向。
      - 2、授权服务器根据 ClientID 确认第三方应用的身份，用户决定是否进行授权。
      - 3、用户同意授权，授权服务器转向第1步中提供的回调 URI，并附带上一个授权码和获取令牌的地址作为参数，这是第二次客户端页面转向。
      - 4、第三方应用通过回调地址收到授权码，然后将授权码与自己的 ClientSecret 一起作为参数，向授权服务器提供的获取令牌的服务地址发起请求，换取令牌。要求服务器的地址应与注册时提供的域名处于同一个域中。
      - 5、授权服务器核对授权码和 ClientSecret，向第三方应用授予令牌。令牌中必定要有的是访问令牌，可选的是刷新令牌。访问令牌用于到资源服务器获取资源，有效期较短，刷新令牌用于在访问令牌失效后重新获取，有效期较长。
      - 6、资源服务器根据访问令牌所允许的权限，向第三方应用提供资源。
    - **不同情况的应对**
      - **其他应用冒充第三方应用骗取授权：**ClientID是可以完全公开的，但ClientSecret 应当只有应用自己才知道。发放令牌时必须能够提供ClientSecret才能成功完成。
      - **先发放授权码，再用授权码换令牌：**客户端转向对于用户是可见的，授权码可能会暴露，但由于并没有 ClientSecret也无法换取到令牌，避免了令牌在传输转向过程中被泄漏的风险。
      - **不能直接把访问令牌的时间调长：**通常访问令牌一旦发放，除非超过了令牌中的有效期，否则很难有其他方式让它失效，所以访问令牌的时效性一般设计的比较短，如果还需要继续用，那就定期用刷新令牌去更新。
    - **缺陷：**虽然最严谨，但要求第三方应用必须有应用服务器向授权服务器获取令牌。
  - **隐式授权模式**
    - **与授权码模式区别：**省略掉了通过授权码换取令牌的步骤，第三方应用不需要服务端支持。同时，由于缺乏ClientSecret的保存地，授权服务器不再去验证第三方应用身份。但还会限制第三方应用的回调 URI 地址必须与注册时提供的域名一致。
    - **安全考虑：**要求第三方应用在注册时提供的回调域名与接受令牌的服务处于同一个域内；在隐式模式中明确禁止发放刷新令牌；令牌必须通过 Fragment 带回，Fragment 是不会跟随请求被发送到服务端的，尽最大努力地避免了令牌从操作代理到第三方服务之间的链路存在被攻击而泄漏出去的可能性。
    - **缺陷：**认证服务器到操作代理之间链路的安全，则只能通过 TLS即 HTTPS来保证中间不会受到攻击了，但无法要求第三方应用同样都支持 HTTPS。
  - **密码模式**

        - **与前两种授权模式区别：**授权码模式和隐私模式属于纯粹的授权模式，与认证没有直接的联系。在密码模式里，认证和授权被整合成了同一个过程，第三方应用拿着用户名和密码直接向授权服务器换令牌。
        - **使用场景：**仅限于用户对第三方应用是高度可信任的场景中使用，或者把第三方视作是系统中与授权服务器相对独立的子模块，在物理上独立于授权服务器部署，但是在逻辑上与授权服务器仍同属一个系统。
    - **客户端模式**
        - **定义：**指第三方应用以自己的名义，向授权服务器申请资源许可。通常用于管理操作或者自动处理类型的场景中。微服务架构并不提倡同一个系统的各服务间有默认的信任关系，客户端模式便是一种常用的服务间认证授权的解决方案。
        - **授权过程：**OAuth2 中还有一种与客户端模式类似的授权模式，即设备码模式。设备从授权服务器获取URI地址和用户码，到验证 URI 中输入用户码。设备会一直循环尝试获取令牌，直到拿到令牌或者用户码过期为止。
- **凭证**
    - **定义：**系统如何保证它与用户之间的承诺是双方当时真实意图的体现，是准确、完整且不可抵赖的？
    - **背景：**对待共享状态信息有两种思路，状态维护在服务端还是客户端？以 HTTP 协议的 Cookie-Session 机制为代表的服务端状态存储在三十年来都是主流的解决方案。不过由于分布式系统中共享数据必然会受到 CAP 不兼容的限制，使得客户端状态存储重新得到关注，如之前只在多方系统中采用的 JWT 令牌方案。
    - **Cookie-Session**
        - **背景：**HTTP为了让服务器有办法区分出发送请求的用户是谁，定义了 HTTP 的状态管理机制，在 HTTP 协议中增加了 Set-Cookie 指令，以键值对的方式向客户端发送一组信息，在此后一段时间内的每次 HTTP 请求中，以名为 Cookie Header 附带着重新发回给服务端，以便服务端区分来自不同客户端的请求。
        - **Set-Cookie: id=icyfenix; Expires=Wed, 21 Feb 2020 07:28:00 GMT; Secure; HttpOnly**

          GET /index.html HTTP/2.0
          Host: icyfenix.cn
          Cookie: id=icyfenix
        - **过程：**系统会把状态信息保存在服务端，在 Cookie 里只传输一个无字面意义的、不重复的字符串，习惯上以sessionid/jsessionid为名，服务器在内存中以 Key/Entity 的结构存储每一个在线用户的上下文状态，使用如超时清理等管理措施。
        - **优点**
            - 状态信息都存储于服务器，只要依靠客户端的同源策略和 HTTPS 的传输层安全，保证 Cookie 中的键值不被窃取而出现被冒认身份的情况，就能完全规避掉上下文信息在传输过程中被泄漏和篡改的风险。
            - 服务端有主动的状态管理能力，可根据自己的意愿随时修改、清除任意上下文信息，譬如很轻易就能实现强制某用户下线的这样功能。
        - **单体服务到多节点方案**
            - **牺牲集群一致性：**让均衡器采用亲和式的负载均衡算法，如根据用户 IP 或者 Session 来分配节点，特定用户发出的请求都被分配到固定节点，每个节点都不重复地保存着一部分用户的状态，若这个节点崩溃了，里面的用户状态便完全丢失。
            - **牺牲集群可用性：**让各个节点之间采用复制式的 Session，每一个节点中的 Session 变动都会发送到组播地址的其他服务器上，但 Session 之间组播复制的同步代价高昂，节点越多时，同步成本越高。
            - **牺牲集群分区容忍性：**让普通的服务节点中不再保留状态，将上下文集中放在一个所有服务节点都能访问到的数据节点中进行存储。此时的矛盾是数据节点就成为了单点，一旦数据节点损坏或出现网络分区，整个集群都不再能提供服务。
    - **JWT**
        - **背景：**当服务器存在多个时，把状态信息存储在客户端，每次随着请求发回服务器去。缺点是无法携带大量信息，有泄漏和篡改的风险。前者不好解决，但使用JWT确保信息不被中间人篡改则还是可以实现的。常见的使用方式是附在名为 Authorization的 Header 发送给服务端。
        - **JWT组成部分**
            - **令牌头Header：**包括令牌的类型，统一为typ:JWT以及令牌签名的算法
            - **负载Payload：**真正向服务端传递的信息。针对认证问题，至少应该包含用户信息，针对授权问题，至少应该包含用户拥有什么角色/权限的信息。
            - **签名Signature：**使用在对象头中公开的特定签名算法，通过特定的密钥Secret，由服务器进行保密，对前面两部分内容进行加密计算。确保负载中的信息是可信的、没有被篡改的。
        - **散列消息认证码HMAC：**带有密钥的哈希摘要算法，不仅保证了内容未被篡改过，还保证了该哈希确实是由密钥的持有人所生成的。
        - **优点**
            - 不需要任何一个服务节点保留任何一点状态信息，就能够保障认证服务与用户之间的承诺是双方当时真实意图的体现，是准确、完整、不可篡改、且不可抵赖的。
            - 本身可以携带少量信息，有利于 RESTful API 的设计，较容易地做成无状态服务，在做水平扩展时就不需要像前面 Cookie-Session 方案那样考虑如何部署的问题
        - **缺陷**
            - **令牌难以主动失效：**JWT 令牌一旦签发，理论上就和认证服务器再没有什么瓜葛了，在到期之前就会始终有效。服务器使用管理策略较困难。
            - **相对更容易遭受重放攻击：**Cookie-Session 也是有重放攻击问题的，只是因为 Session 中的数据控制在服务端手上，应对重放攻击会相对主动一些。
            - **只能携带相当有限的数据：**在令牌中存储过多的数据不仅耗费传输带宽，还有额外的出错风险。
            - **必须考虑令牌在客户端如何存储**
            - **无状态也不总是好的**
- **保密**
    - **定义：**系统如何保证敏感数据无法被包括系统管理员在内内外部人员所窃取滥用？
    - **保密的强度**
        - **以摘要代替明文：**如果密码本身比较复杂，简单的哈希摘要至少可以保证即使传输过程中有信息泄漏，也不会被逆推出原信息；即使密码在一个系统中泄漏了，也不至于威胁到其他系统的使用，但这种处理不能防止弱密码被彩虹表攻击所破解。

- **先加盐值再做哈希：**是应对弱密码的常用方法，盐值可以替弱密码建立一道防御屏障，一定程度上防御已有的彩虹表攻击，但并不能阻止加密结果被监听、窃取后，攻击者直接发送加密结果给服务端进行冒认。
- **将盐值变为动态值能有效防止冒认：**如果每次传输时都掺入了动态盐值，让加密的结果都不同，那即使被窃取了，也不能冒用来进行另一次调用。在双方通信均可能泄漏的前提下协商出只有双方才知道的保密信息是可行的，但协商出盐值的过程将变得复杂，而且每次协商只保护一次操作，也难以阻止对其他服务的重放攻击。
- **给服务加入动态令牌：**在网关或其他流量公共位置建立校验逻辑，服务端愿意付出在集群中分发令牌信息等代价的前提下，可以做到防止重放攻击，但是依然不能抵御传输过程中被嗅探而泄漏信息的问题。
- **启用 HTTPS 可以防御链路上的恶意嗅探：**也能在通信层面解决了重放攻击的问题。但依然有因客户端被攻破产生伪造根证书、有因服务端被攻破产生的证书泄漏而被中间人冒认、有因CRL更新不及时或者OCSP Soft-fail 产生吊销证书被冒用、有因 TLS 的版本过低或密码学套件选用不当产生加密强度不足的风险。
- **为了抵御上述风险，保密强度还要进一步提升：**譬如银行会使用独立于客户端的存储证书的物理设备（俗称的 U 盾）来避免根证书被恶意程序窃取伪造；涉及到账号、金钱等操作时，会使用双重验证开辟一条独立于网络的信息通道如手机验证码来显著提高冒认的难度；甚至一些关键企业（如国家电网）或机构（如军事机构）会专门建设遍布全国各地的与公网物理隔离的专用内部网络来保障通信安全。

- **密码存储和验证**
  - **普通安全强度**
    密码创建过程
    - 1、用户在客户端注册，输入明文密码：123456
    - 2、客户端对用户密码进行简单哈希摘要，可选的算法有 MD2/4/5、SHA1/256等
    - 3、为了防御彩虹表攻击应加盐处理，客户端加盐只取固定的字符串即可，最多用伪动态的盐值，即服务端不需要额外通信可以得到的信息。
    - 4、假设攻击者截获了信息，得到了摘要结果和盐值，就可以枚举遍历弱密码，然后对每个密码再加盐计算，就得到一个针对固定盐值的对照彩虹表。为了应对这种暴力破解，并不提倡在盐值上做动态化，更理想的方式是引入慢哈希函数（函数执行时间是可以调节的哈希函数，通常是以控制调用次数来实现的。）来解决。
    - 5、只需防御被拖库后针对固定盐值的批量彩虹表攻击，为每一个密码即客户端传来的哈希值产生一个随机的盐值。常用密码学安全伪随机数生成器CSPRNG来生成一个长度与哈希值长度相等的随机字符串。
    - 6、将动态盐值混入客户端传来的哈希值再做一次哈希，产生出最终的密文，并和上一步随机生成的盐值一起写入到同一条数据库记录中。由于慢哈希算法占用大量处理器资源，并不推荐在服务端中采用。
  - **密码验证过程**
    - 1、客户端用户在登录页面中输入密码明文，经过与注册相同的加密过程，向服务端传输加密后的结果。（与创建过程第3或第4步生成的结果相同）。
    - 2、服务端接受到哈希值，从数据库中取出登录用户对应的密文和盐值，采用相同的哈希算法，对客户端传来的哈希值、服务端存储的盐值计算摘要结果。
    - 3、比较上一步的结果和数据库储存的哈希值是否相同
- **客户端加密**
  - **背景：**为了保证信息不被黑客窃取而做客户端加密没有太多意义，对绝大多数的信息系统来说，启用 HTTPS 可以说是唯一的实际可行的方案。但是为了保证密码不在服务端被滥用，在客户端就开始加密是很有意义的，例如避免被拖库等。
  - **客户端加密对防御泄密会没有意义：**原因是网络通信并非由发送方和接收方点对点进行的，客户端无法决定用户送出的信息能不能到达服务端，或者会经过怎样的路径到达服务端，在传输链路必定是不安全的假设前提下，无论客户端做什么防御措施，最终都会沦为“马其诺防线”。
  - 在客户端加盐传输的做法通常都得不偿失，客户端无论是否动态加盐，都不可能代替 HTTPS。真正防御性的密码加密存储确实应该在服务端中进行，但是为了防御服务端被攻破而批量泄漏密码的风险，并不是为了增加传输过程的安全。

传输

- **定义：**系统如何保证通过网络传输的信息无法被第三方窃听、篡改和冒充?
- **摘要、加密与签名**
  - **摘要：**也称之为数字摘要或数字指纹。信息通过哈希算法计算出来摘要值，理想的哈希算法都具备易变性和不可逆性。可以在源信息不泄漏的前提下辨别其真伪。
  - **加密：**加密是可逆的，现代密码学安全建立在特定问题的计算复杂度之上。
    - **对称加密算法：**为保证两两通信都采用独立的密钥，密钥数量与成员数量平方成正比；最关键的是如何才能将一个只能让通信双方才能知道的密钥传输给对方。若能保证传输安全，也就没加密算法什么事了。
    - **非对称加密算法：**公钥加密，私钥解密，这种就是**加密，用于向私钥所有者发送信息**，这个信息可能被他人篡改，但是无法被他人得知；私钥加密，公钥解密，这种就是**签名，用于让所有公钥所有者验证私钥所有者的身份**，并且用来防止私钥所有者发布的内容被篡改。但是不用来保证内容不被他人获得。
    - **安全保密：**如果甲想给乙发一个安全保密的数据，那么应该甲乙各自有一个私钥，甲先用乙的公钥加密这段数据，再用自己的私钥加密这段加密后的数据。最后再发给乙，这样确保了内容即不会被读取，也不能被篡改。
    - **非对称加密算法缺陷：**计算复杂度都相当高，性能比对称加密要有数量级差距，因此难以支持分组，所以主流的非对称加密算法都只能加密不超过密钥长度的数据，这决定了非对称加密不能直接用于大量数据的加密。
    - **密码学套件：**用非对称加密来安全地传递少量数据给通信的另一方，然后再以这些数据为密钥即“密钥协商”，采用对称加密来安全高效地大量加密传输数据。
  - **JWT 令牌的签名不能保证负载中的信息不可篡改、不可抵赖：**即使有了哈希摘要、对称和非对称加密，但公钥在网络传输过程中可能已经被篡改，如果获取公钥的网络请求被攻击者截获并篡改，返回了攻击者自己的公钥，那以后攻击者就可以用自己的私钥来签名，让资源服务器无条件信任它的所有行为了。
- **数字证书**
  - **现实世界达成信任的方法：**基于共同私密信息的信任和基于权威公证人的信任
  - **网络世界达成信任的方法：**由于不能假设授权服务器和资源服务器是互相认识的，因此采用基于权威公证人的信任方式，即目前标准的保证公钥可信分发的公开密钥基础设施PKI。
  - **数字证书认证中心CA：**承担公钥体系中公钥的合法性检验的责任，且一般预置系统中或可安装，避免伪装CA。CA负责签发证书 Certificate，是对特定公钥信息的一种公证载体，是权威 CA 对特定公钥未被篡改的签名背书。

- **传输安全层**
  - **背景：**如果从确定加密算法、生成密钥、公钥分发、CA 认证、核验公钥、签名、验证，每一个步骤都要由最终用户来完成的话，将是十分繁琐且极难推广的。最合理的做法就是在传输层之上、应用层之下加入专门的安全层来实现，这样对上层原本基于 HTTP 的 Web 应用来说影响甚至是无法察觉的，即构建传输安全层。
  - **发展：**SSL 协议（Secure Sockets Layer）->TLS（Transport Layer Security）
  - **TLS1.2**
    - **意义：**传输安全层是保障所有信息都是第三方无法窃听（加密传输）、无法篡改（一旦篡改通信算法会立刻发现）、无法冒充（证书验证身份）的。
    - **过程**
      - **客户端请求Client Hello：**客户端向服务器请求进行加密通信，以明文的形式，向服务端提供支持的协议版本、生成的 32 Bytes 随机数（用于产生加密的密钥）、SessionID（用于复用TLS连接）、支持的密码学套件和数据压缩算法、扩展信息。
      - **服务器回应Server Hello：**服务器接收到客户端的通信请求后，支持的协议版本和加密算法组合与客户端相匹配的话，向客户端发出回应，包括服务端确认使用的 TLS 协议版本、第二个 32 Bytes 的随机数（用于产生加密的密钥）、SessionID、列表中选定的密码学算法套件、列表中选定的数据压缩方法、扩展信息。如果协商出的加密算法组合是依赖证书认证的，服务端还要发送出自己的 X.509 证书。
      - **客户端确认Client Handshake Finished：**客户端收到服务器应答后，验证服务器的证书合法性，从证书中取出服务器的公钥，并向服务器发送客户端证书（可选，部分服务端并不是面向全公众）、第三个 32 Bytes 的随机数（以服务端传过来的公钥加密）将与前两次发送的随机数一起，根据特定算法后续内容传输时的对称加密算法所采用的私钥等。
      - **服务端确认Server Handshake Finished：**服务端向客户端回应最后的确认通知，包括编码改变通知，表示随后的信息都将用双方商定的加密方法和密钥发送。
    - **HTTPS：**建立在安全传输层之上的 HTTP 协议，就被称为“HTTP over SSL/TLS”。采用不同的协议版本、不同的密码学套件、证书是否有效、服务端/客户端对面对无效证书时的处理策略如何都导致了不同 HTTPS 站点的安全强度的不同。

- **验证**
  - **定义：**系统如何确保提交到每项服务中的数据是合乎规则的，不会对系统稳定性、数据一致性、正确性产生风险？
  - 在 Java 里有验证的标准做法，提倡的做法是把校验行为从分层中剥离出来，不是在哪一层做，而是在 Bean 上做。即 Java Bean Validation。