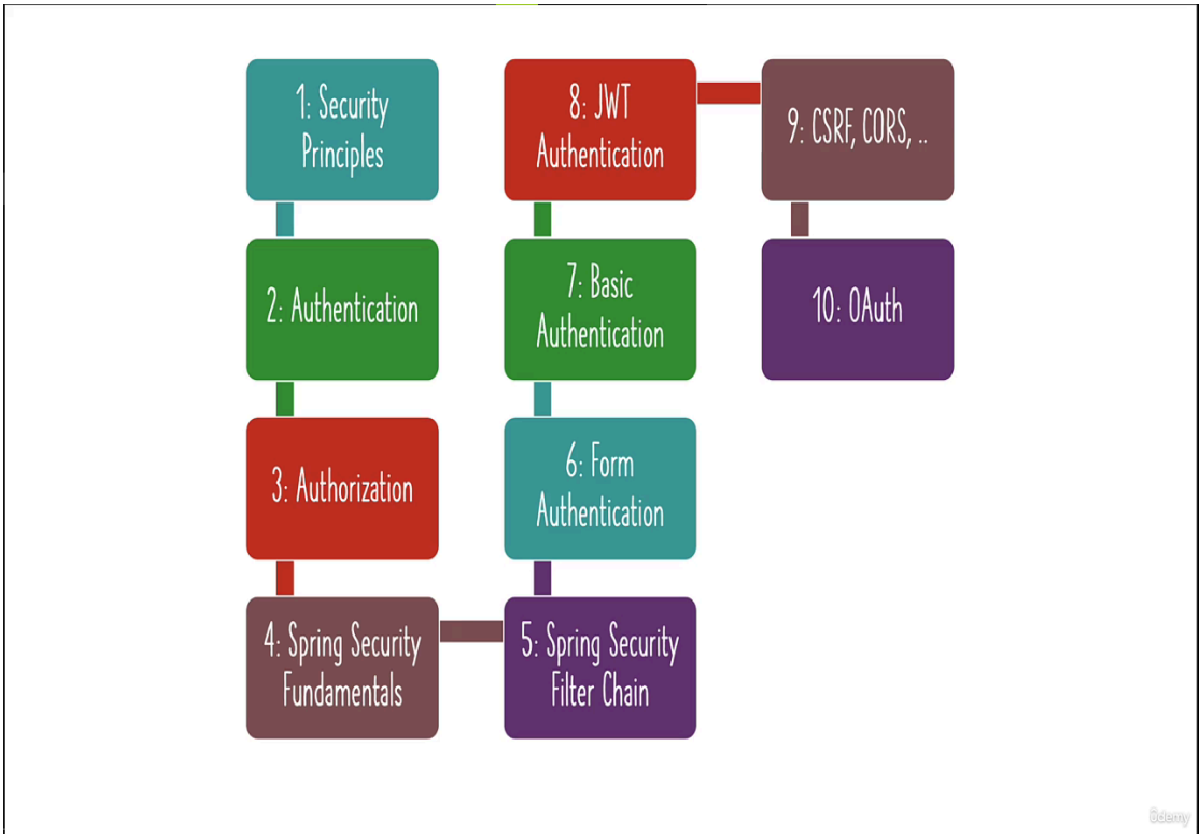
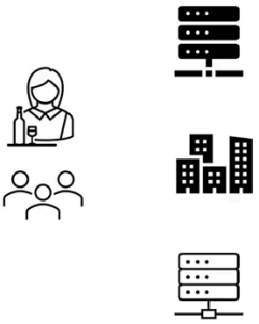
Spring security



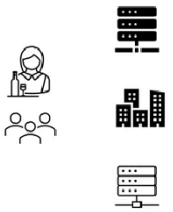
Understanding Security Fundamentals

- In any **system:**
  - You have **resources**
    - A REST API, A Web Application, A Database, A resource in the cloud, ...
  - You have **identities**
    - Identities need to access to resources and perform actions
      - For example: Execute a REST API call, Read/modify data in a database
  - Key Questions:
    - How to **identify users?**
    - How to configure resources they can access & actions that are allowed?
- **Authentication** (is it the right user?)
  - UserId/password (What do you remember?)
  - Biometrics (What do you possess?)
- **Authorization** (do they have the right access?)
  - User XYZ can only read data
  - User ABC can read and update data



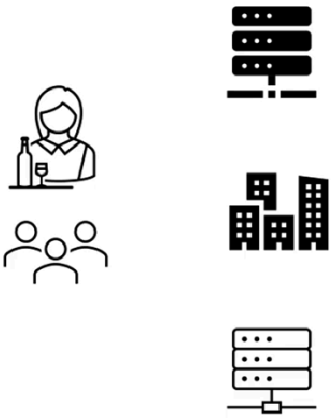
Understanding Important Security Principles

- A chain is **only as strong** as its WEAKEST link
  - Small security flaw makes an app with robust architecture vulnerable
- **6 Principles** Of Building Secure Systems
  - **1:** Trust Nothing
    - Validate every request
    - Validate piece of data or information that comes into the system
  - **2:** Assign Least Privileges
    - Start the design of the system with security requirements in mind
    - Have a clear picture of the user roles and accesses
    - Assign Minimum Possible Privileges at all levels
      - Application
      - Infrastructure (database + server + ..)
  - **3:** Have Complete Mediation
    - How were Medieval Fort's protected?
      - Everyone had to pass through one main gate
    - Apply a well-implemented security filter. Test the role and access of each user.



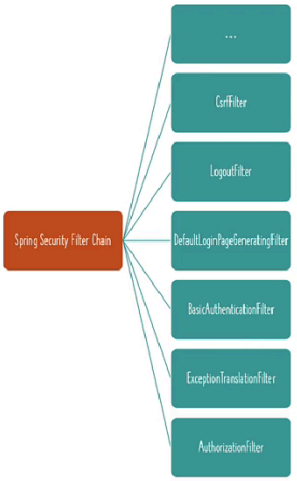
Understanding Important Security Principles

- **4:** Have Defense In Depth
  - Multiple levels of security
    - Transport, Network, Infrastructure
    - Operating System, Application, ..
- **5:** Have Economy Of Mechanism
  - Security architecture should be simple
  - Simple systems are easier to protect
- **6:** Ensure Openness Of Design
  - Easier to identify and fix security flaws



How does Spring Security Work? (2)

- Spring Security executes a series of filters
  - Filters **provide these features**:
    - **Authentication**: Is it a valid user? (Ex: BasicAuthenticationFilter)
    - **Authorization**: Does the user have right access?(Ex: AuthorizationFilter)
    - **Other Features**:
      - Cross-Origin Resource Sharing (CORS) - CorsFilter
        - Should you allow AJAX calls from other domains?
      - Cross Site Request Forgery (CSRF) - CsrfFilter
        - A malicious website making use of previous authentication on your website
        - Default: CSRF protection enabled for update requests - POST, PUT etc..
      - Login Page, Logout Page
        - LogoutFilter, DefaultLoginPageGeneratingFilter, DefaultLogoutPageGeneratingFilter
      - Translating exceptions into proper Http Responses (ExceptionTranslationFilter)
  - **Order of filters** is important (typical order shown below)
    - **1:** Basic Check Filters - CORS, CSRF, ..
    - **2:** Authentication Filters
    - **3:** Authorization Filters



CSRF

Getting started with Cross-Site Request Forgery (CSRF)

- **1:** You are logged-in to your bank website
  - A cookie `Cookie-A` is saved in the your web browser
- **2:** You go to a malicious website without logging out
- **3:** Malicious website executes a bank transfer without your knowledge using Cookie-A
- How can you **protect from CSRF**?
  - **1**: Synchronizer token pattern
    - A token created for each request
    - To make an update (POST, PUT, ..), you need a CSRF token from the previous request
  - **2**: SameSite cookie (`Set-Cookie: SameSite=Strict`)
    - application.properties
      - `server.servlet.session.cookie.same-site=strict`

Why csrf()

https://stackoverflow.com/questions/52363487/what-is-the-reason-to-disable-csrf-in-spring-boot-web-application

Getting Started with CORS

```
@Bean
public WebMvcConfigurer corsConfigurer() {
    return new WebMvcConfigurer() {
        public void addCorsMappings(CorsRegistry registry) {
            registry.addMapping("/**")
                .allowedMethods("*")
                .allowedOrigins("http://localhost:3000");
        }
    };
}
```

- Browsers do NOT allow AJAX calls to resources outside current origin
- Cross-Origin Resource Sharing (CORS): Specification that allows you to configure which cross-domain requests are allowed
  - **Global Configuration**
    - Configure `addCorsMappings` callback method in `WebMvcConfigurer`
  - **Local Configuration**
    - `@CrossOrigin` - Allow from all origins
    - `@CrossOrigin(origins = "https://www.in28minutes.com")` - Allow from specific origin

Storing User Credentials

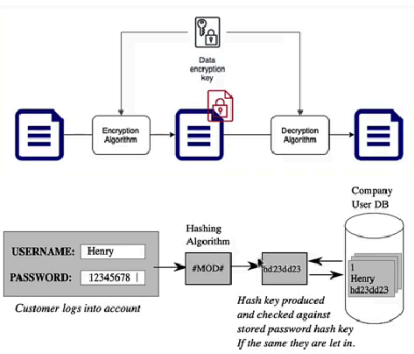
```
@Bean
public UserDetailsService userDetailsService(DataSource dataSource) {
    UserDetails user = User.builder()
        .username("in28minutes")
        //.password("{noop}dummy")
        .password("dummy")
        .roles("USER")
        .passwordEncoder(str -> passwordEncoder().encode(str))
        .build();
    JdbcUserDetailsManager users = new JdbcUserDetailsManager(dataSource);
    users.createUser(user);
    return users;
    //return new InMemoryUserDetailsManager(user);
}
```

- User credentials can be stored in:
  - **In Memory** - For test purposes. Not recommended for production.
  - **Database** - You can use JDBC/JPA to access the credentials.
  - **LDAP** - Lightweight Directory Access Protocol
    - Open protocol for directory services and authentication

Encoding vs Hashing vs Encryption

- **Encoding**: Transform data - one form to another
  - Does NOT use a key or password
  - Is reversible
  - Typically NOT used for securing data
  - Usecases: Compression, Streaming
  - Example: Base 64, Wav, MP3
- **Hashing**: Convert data into a Hash (a String)
  - One-way process
  - NOT reversible
    - You CANNOT get the original data back!
  - Usecases: Validate integrity of data
  - Example: bcrypt, scrypt
- **Encryption**: Encoding data using a key or password
  - You need to key or password to decrypt
  - Example: RSA

Security



https://upload.wikimedia.org/wikipedia/commons/5/5e/CPT-Hashing-Password-Login.svg

Spring Security - Storing Passwords

- Hashes like SHA-256 are no longer secure
- Modern systems can perform billions of hash calculations a second
  - AND systems improve with time!
- Recommended: Use adaptive one way functions with Work factor of 1 second
  - It should take at least 1 second to verify a password on your system
  - Examples: bcrypt, scrypt, argon2, ..
- PasswordEncoder - interface for performing one way transformation of a password
  - (REMEMBER) Confusingly Named!
  - BCryptPasswordEncoder

JWT

Getting Started with JWT

- **Basic Authentication**
  - No Expiration Time
  - No User Details
  - Easily Decoded
- How about a **custom token system**?
  - Custom Structure
  - Possible Security Flaws
  - Service Provider & Service Consumer should understand
- **JWT (Json Web Token)**
  - Open, industry standard for representing claims securely between two parties
  - Can Contain User Details and Authorizations

```
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.ey
JzdWIiOiIxMjM0NTY3ODkwIiwibmFtZSI6Ikpva
G4gRG9lIiwiaWF0IjoxNTE2MjM5MDIyfQ.SflKx
wRJSMeKKF2QT4fwpMeJf36POk6yJV_adQssw5c
```

What does a JWT contain?

- **Header**
  - Type: JWT
  - Hashing Algorithm: HS512
- **Payload**
  - **Standard Attributes**
    - **iss:** The issuer
    - **sub:** The subject
    - **aud:** The audience
    - **exp:** When does token expire?
    - **iat:** When was token issued?
  - **Custom Attributes**
    - **youratt1:** Your custom attribute 1
- **Signature**
  - Includes a Secret

HEADER: ALGORITHM & TOKEN TYPE

```
{
  "alg": "HS256",
  "typ": "JWT"
}
```

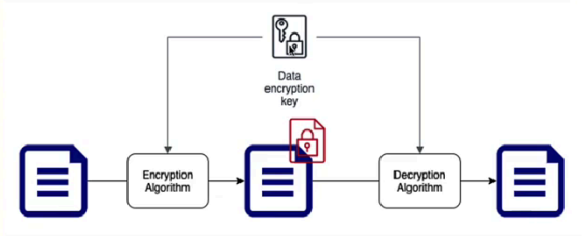
PAYLOAD: DATA

```
{
  "sub": "1234567890",
  "name": "John Doe",
  "iat": 1516239022
}
```

VERIFY SIGNATURE

```
HMACSHA256(
  base64UrlEncode(header) + "." +
  base64UrlEncode(payload),
  your-256-bit-secret
) secret base64 encoded
```

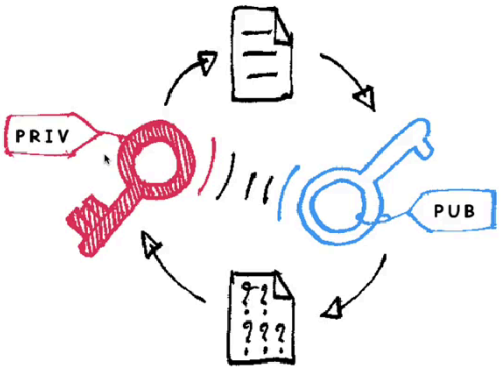
Symmetric Key Encryption



- Symmetric encryption algorithms use the **same key for encryption and decryption**
- Key Factor 1: Choose the **right encryption algorithm**
- Key Factor 2: How do we **secure the encryption key**?
- Key Factor 3: How do we **share the encryption key**?

Asymmetric Key Encryption

- **Two Keys :** Public Key and Private Key
- Also called **Public Key Cyptography**
- Encrypt data with Public Key and decrypt with Private Key
- Share Public Key with everybody and keep the Private Key with you(YEAH, ITS PRIVATE!)
- No crazy questions:
  - Will somebody not figure out private key using the public key?
- **Best Practice**: Use Asymmetric Keys



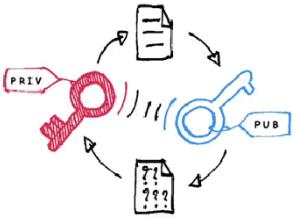
https://commons.wikimedia.org/wiki/File:Asymmetric_encryption_(co

Understanding High Level JWT Flow

- **1**: Create a JWT
  - Needs Encoding
    - **1**: User credentials
    - **2**: User data (payload)
    - **3**: RSA key pair
  - We will create a JWT Resource for creating JWT later
- **2**: Send JWT as part of request header
  - Authorization Header
  - Bearer Token
  - `Authorization: Bearer ${JWT_TOKEN}`
- **3**: JWT is verified
  - Needs Decoding
  - RSA key pair (Public Key)

Getting Started with JWT Security Configuration

- JWT Authentication using Spring Boot's OAuth2 Resource Server
  - **1**: Create Key Pair
    - We will use `java.security.KeyPairGenerator`
    - You can use openssl as well
  - **2**: Create RSA Key object using Key Pair
    - `com.nimbusds.jose.jwk.RSAKey`
  - **3**: Create JWKSource (JSON Web Key source)
    - Create `JWKSet` (a new JSON Web Key set) with the RSA Key
    - Create `JWKSource` using the `JWKSet`
  - **4**: Use RSA Public Key for Decoding
    - `NimbusJwtDecoder.withPublicKey(rsaKey().toRSAPublicKey()).build()`
  - **5**: Use JWKSource for Encoding
    - `return new NimbusJwtEncoder(jwkSource());`



Getting Started with JWT Resource

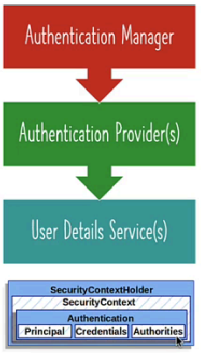
```
username:"in28minutes",
password:"dummy"

Response
{
"token": "TOKEN_VALUE"
}
```

- Step 1: Use Basic Auth for getting the JWT Token
- Step 2-n: Use JWT token as Bearer Token for authenticating requests

Spring security Authentication

Understanding Spring Security Authentication

- Authentication is done as part of the Spring Security Filter Chain!
- 1: **AuthenticationManager** - Responsible for authentication
  - Can interact with multiple authentication providers
- 2: **AuthenticationProvider** - Perform specific authentication type
  - `JwtAuthenticationProvider` - JWT Authentication
- 3: **UserDetailsService** - Core interface to load user data
- How is authentication result stored?
  - SecurityContextHolder > SecurityContext > Authentication > GrantedAuthority
    - **Authentication** - (After authentication) Holds user (**Principal**) details
    - **GrantedAuthority** - An authority granted to principal (roles, scopes,..)



Spring security Authorization

Exploring Spring Security Authorization

- **1:** Global Security: authorizeHttpRequests
  - `.requestMatchers("/users").hasRole("USER")`
    - hasRole, hasAuthority, hasAnyAuthority, isAuthenticated
- **2:** Method Security (@EnableMethodSecurity)
  - @Pre and @Post Annotations
    - @PreAuthorize("hasRole('USER') and #username == authentication.name")
    - @PostAuthorize("returnObject.username == 'in28minutes'")
  - JSR-250 annotations
    - @EnableMethodSecurity(jsr250Enabled = true)
    - @RolesAllowed({"ADMIN", "USER"})
  - @Secured annotation
    - @EnableMethodSecurity(securedEnabled = true)
    - @Secured({"ADMIN", "USER"})