Understanding

Spring Security

— Nandakumar Purohit

Spring Security

Agenda

- What is a Security?
- What is Securing Web Applications?
- What layers of an Enterprise Application do you intend to secure?
- Security Mechanisms OR Types of Securities
- What is Spring Security
- Spring Security Core Features
- Spring Security 5 - New Features

Spring Security

What is Security?

- *"Security is a Non-functional requirement implementing which we ensure that our application / services are not accessed by Unauthorized Users"*

Spring Security

What is Securing Web Applications?

- *Implementing a Security for a Web Application involves securing MVC Actions & Web Pages*

Spring Security

What layers of an Enterprise Application do you intend to secure?

- *MVC Actions*
- *Web Pages*
- *REST APIs*
- *Microservices*
- *Data*
- *Database Servers*
- *Confidential Data within Database*
- *Targeted Client Devices*

Spring Security

Security Mechanisms OR Types of Securities

- ???

Spring Security

What is Spring Security

- New Term in the Industry called “DevSecOps”
- Security is a Non-functional requirement

“Spring Security is a powerful and highly customizable authentication and access-control framework. It is a de-facto standard for securing Spring-based applications”

Spring Security

What is Spring Security

- Spring Security has top-level projects to deal with Authentication mechanisms like **LDAP, OAuth & SAML**
- It also gives mechanisms to deal with common **security attacks** like:
  - Session Fixation
  - Clickjacking
  - CSRF
- Spring Security is having very good **integration** with **Spring Framework projects** like:
  - Spring MVC
  - Spring WebFlux
  - Spring Data
  - Spring Boot

Spring Security

What is Spring Security

Spring Security Terminologies	
Principal	Any user, device, or system (application) that would like to interact with your application
Authentication	A process by which your application makes sure that the principal is who they claim to be.
Credentials	When a principal tries to interact with your application, the authentication process kicks in and challenges the principal to pass on some values
Authorization	• After successful authentication, the principal is checked again for actions that it can perform on your application. • This process of checking rights for a principal and then granting necessary permissions is called authorization.
Secured Item / Resource	The item or resource that is marked as secured and requires the principal (user) to successfully complete both authentication and authorization
Granted Authority	A Spring Security object that contains/holds permissions/access-right details of a principal
Security Context	A Spring Security object that holds a principal's authentication details

Spring Security

Core Features of Spring Security

Authentication	Authorization
SAML	Web URL
LDAP	Method Invocation
OAuth/OIDC	Domain Instance
CAS	Web Service
JAAS	

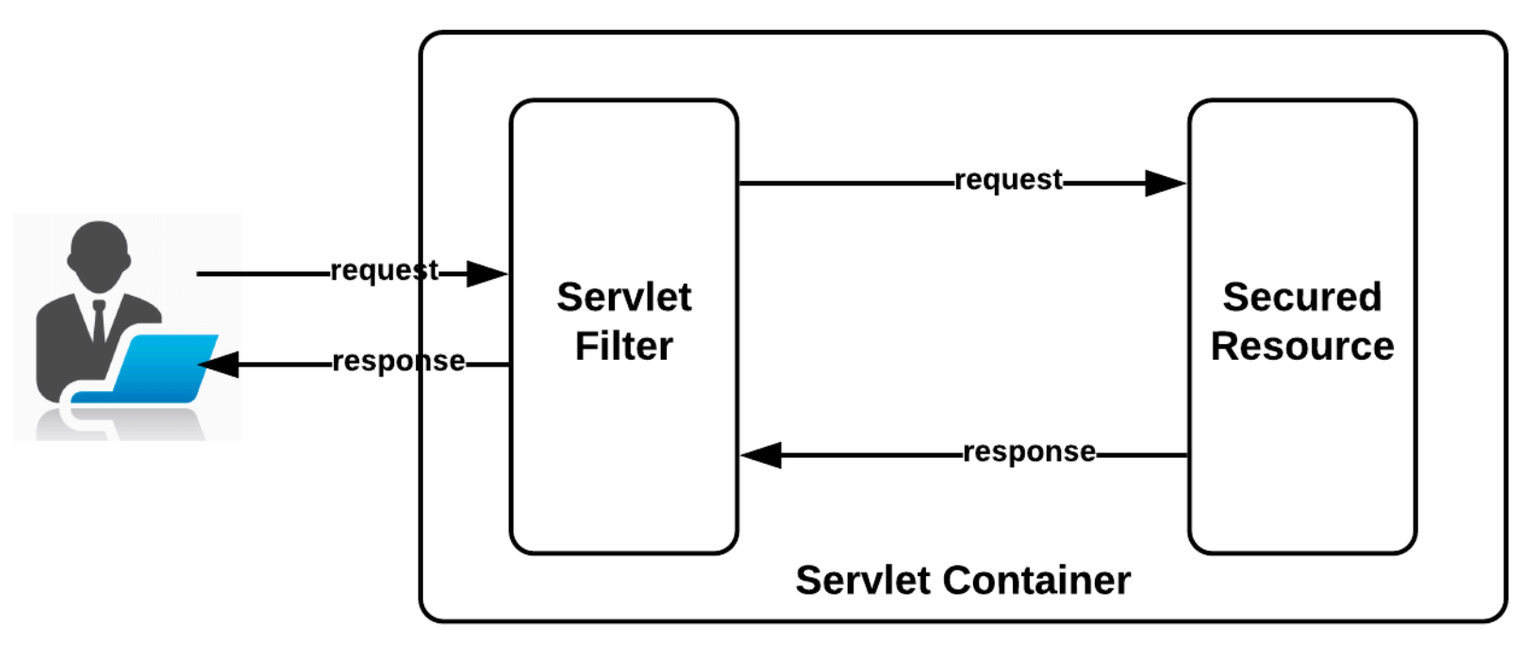
Spring Security

New Features of Spring Security 5

Important New Features in Spring Security 5	
Support for OAuth 2.0 & OpenID Connect 1.0	• Allows users to log in to your application using their existing **OAuth** provider (for example, **GitHub**) or **OIDC** provider (for example, **Google**). • OAuth is implemented using **Authorization Code Flow**.
Reactive Support	• Spring 5 introduced a new reactive web application framework—**Spring WebFlux**. • Spring Security made sure that this web application framework is fully supported in all aspects (authentication and authorization) using reactive concepts.
Improved Password Encoding	• The introduction of the **password-encoding delegation** allows usage of more than one algorithm for encoding various passwords. • The way Spring identifies the algorithm is by reading the **prefix of the encoded password**, which contains the algorithm used to encode the password. • The format is **{algorithm}encoded_password**

Spring Security

Working of Spring Security



Servlet Filter

- It comes before the request reaches the actual resource and also before the response if sent back to the consumer.

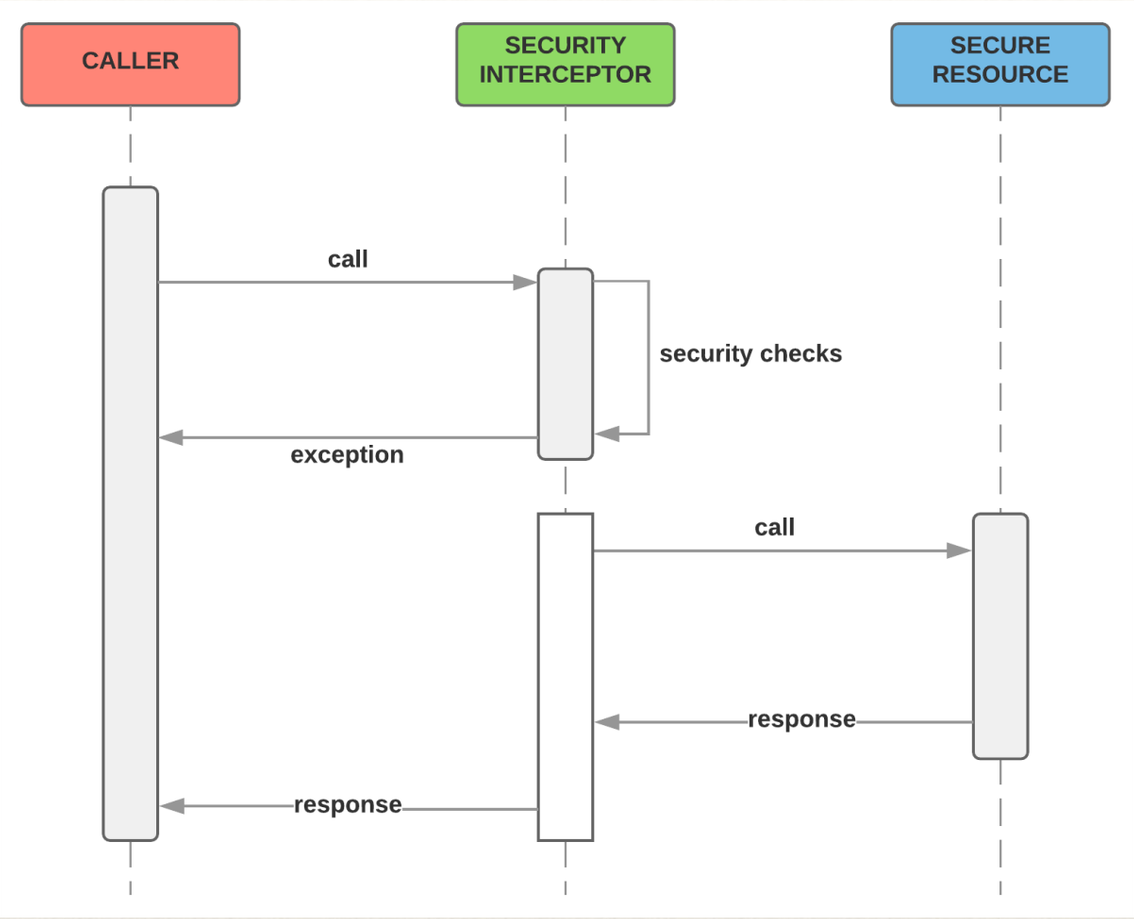
Filter Chain

- You can embed any number of Servlet Filters before they reach the actual resource.
- The filters are fired according to the order in which they are declared in web.xml
- Spring Security works on a number of Servlet Filters arranged as a Filter Chain, each filter performing a single responsibility

Spring Security

Working of Spring Security

Security Interceptor (DelegatingFilterProxy)

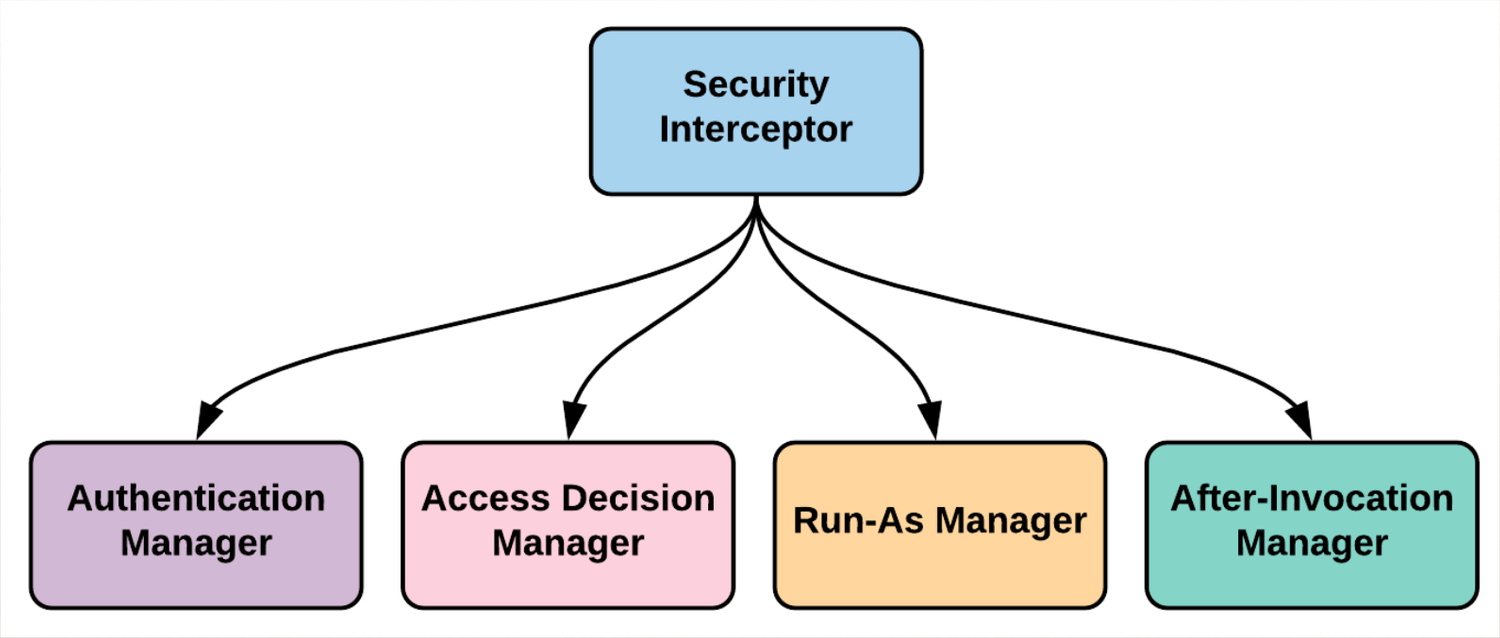


The Security Interceptor makes sure that, according to various security configurations set up for your application, it delegates the work to appropriate parties and makes sure that everyone is happy before actually reaching the resource requested by the caller.

Spring Security

Working of Spring Security

Security Interceptor & Associated Managers

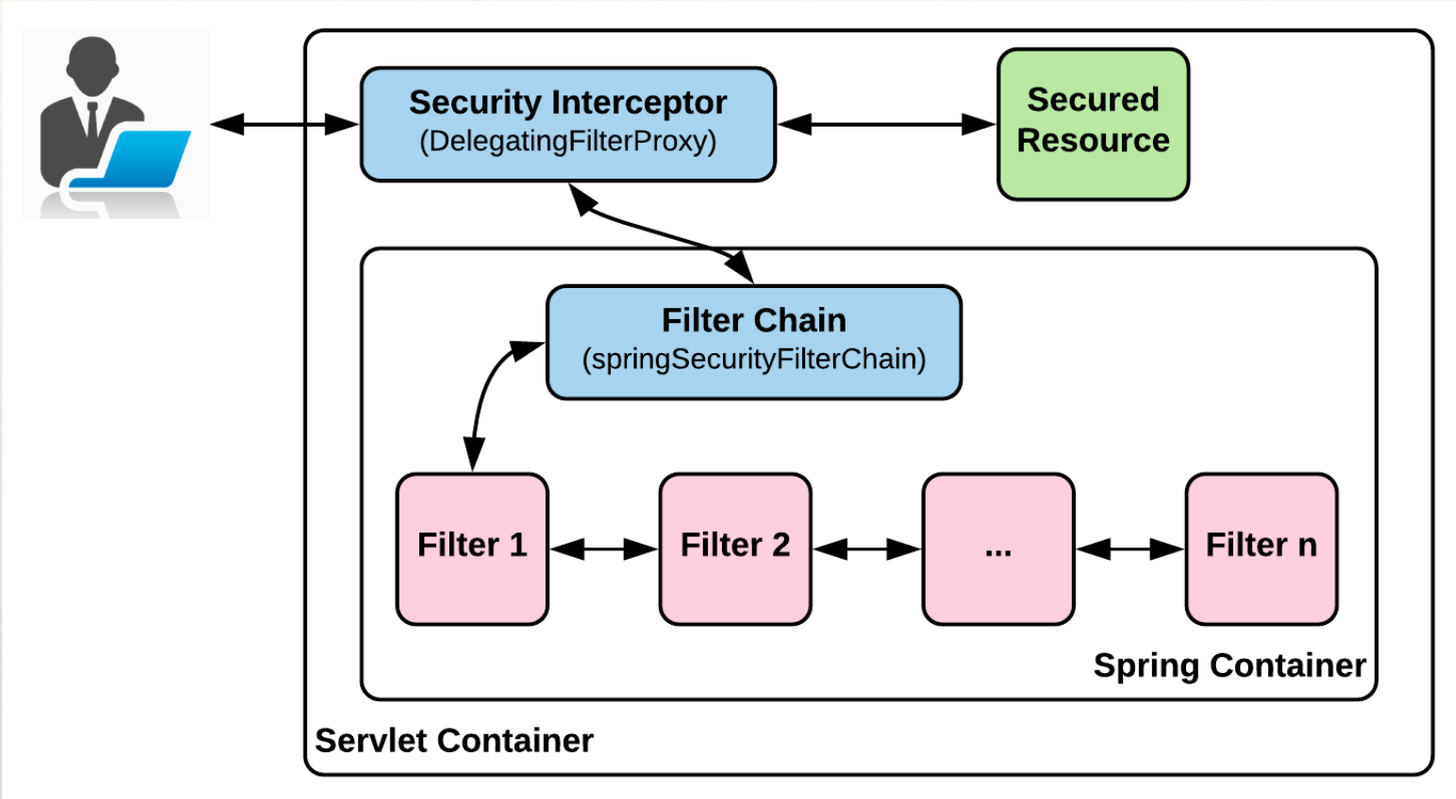


- `DelegatingFilterProxy` is a Servlet Filter that has to be configured in your `web.xml` file, which then delegates to a Spring-managed bean (`@Bean`) that implements a `ServletFilter` interface.
- All the requests to the web application (`/* mapping`) would go through the `DelegatingProxyFilter` filter.
- If you are using the default Spring Security setup, the request would then be received by `FilterChainProxy`.
- `FilterChainProxy` is responsible for passing the request through the various Servlet Filters configured as part of Spring Security.

Spring Security

Working of Spring Security

Security Interceptor & Filters

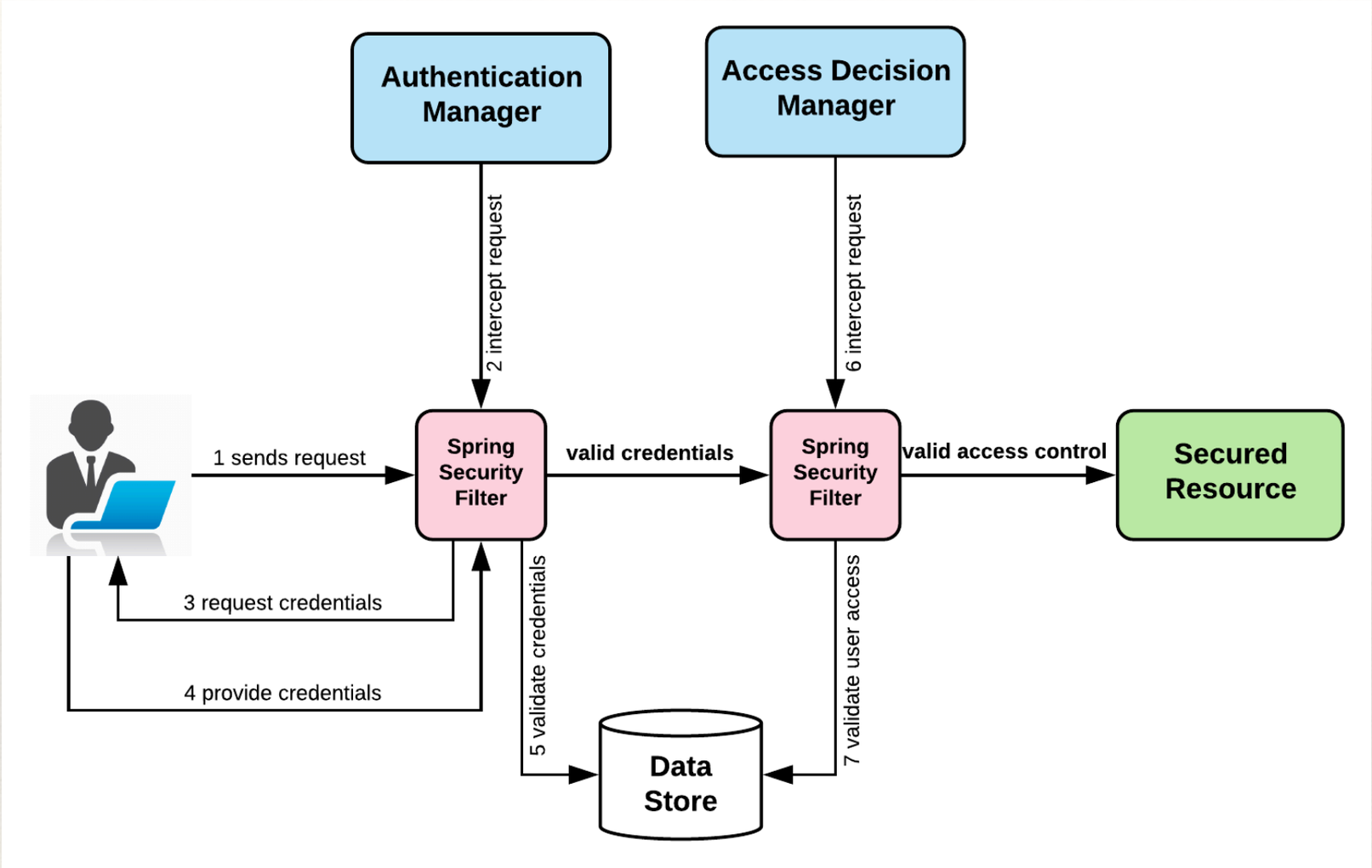


The request from the caller reaches `DelegatingFilterProxy`, which delegates to `FilterChainProxy` (Spring Bean), which in turn passes the request through a number of filters, and after successful execution, grants access to the secured resource the caller has asked for.

Spring Security

Working of Spring Security

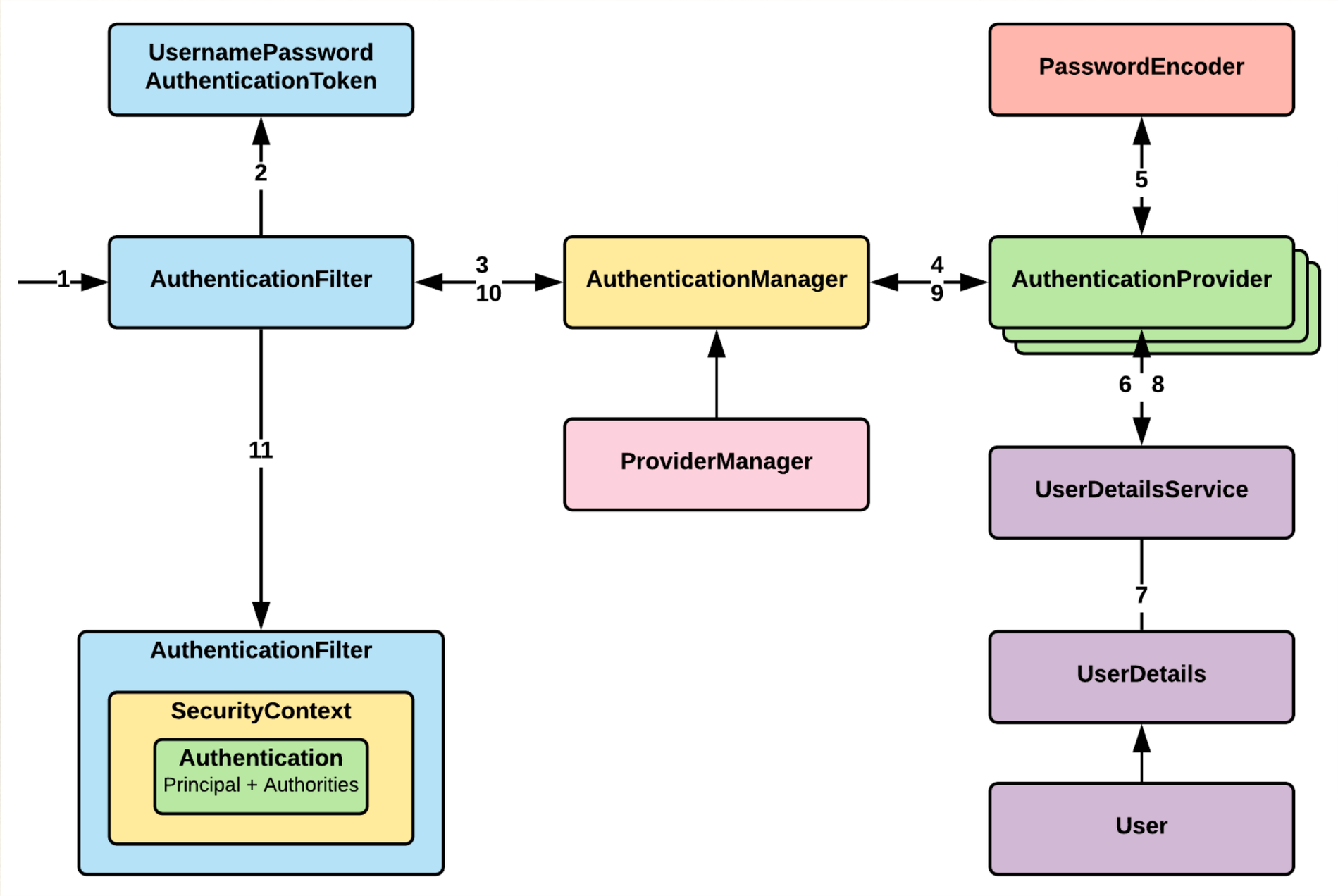
Security Interceptor & Filters



The request from the caller reaches `DelegatingFilterProxy`, which delegates to `FilterChainProxy` (Spring Bean), which in turn passes the request through a number of filters, and after successful execution, grants access to the secured resource the caller has asked for.

Spring Security

Authentication



Spring Security

Authentication

- The Servlet filter code execution in the filter chain keeps skipping until the right filter is reached.
- Once it reaches the right authentication filter based on the authentication mechanism used, it extracts the supplied credentials (most commonly a username and password) from the caller.
- Using the supplied values, the filter (`UsernamePasswordAuthenticationFilter`) creates an `Authentication` object (in the preceding diagram, `UsernamePasswordAuthenticationToken` is created using the username and password supplied in **Step 2**).
- The `Authentication` object created in **Step 2** is then used to call the `authenticate` method in the`AuthenticationManager` interface:

```
public interface AuthenticationManager {
    Authentication authenticate(Authentication authentication)
        throws AuthenticationException;
}
```

Spring Security

Authentication

- The actual implementation is provided by **ProviderManager**, which has a list of configured `AuthenticationProvider`.

```
public interface AuthenticationProvider {
    Authentication authenticate(Authentication authentication)
        throws AuthenticationException;
    boolean supports(Class<?> authentication);
}
```

- The request passes through various providers and, in due course, tries to authenticate the request.
- There are a number of `AuthenticationProvider` as part of Spring Security.
- `UserDetailsService` can be used as one of them

```
public interface UserDetailsService {
    UserDetails loadUserByUsername(String username) throws
        UsernameNotFoundException;
}
```

Spring Security

Authentication

- `UserDetailsService` retrieves `UserDetails` (and implements the `User` interface) using the supplied username.
- If all goes well, Spring Security creates a fully populated `Authentication` object (authenticate: true, granted authority list and username), which will contain various necessary details.
- The `Authentication` object is stored in the `SecurityContext` object by the filter for future use.

The `authenticate` method in `AuthenticationManager` can return the following:

- An `Authentication` object with `authenticated=true`, if Spring Security can validate the supplied user credentials
- An `AuthenticationException`, if Spring Security finds that the supplied user credentials are invalid
- `null`, if Spring Security cannot decide whether it is true or false (confused state)

Spring Security

Setting up AuthenticationManager

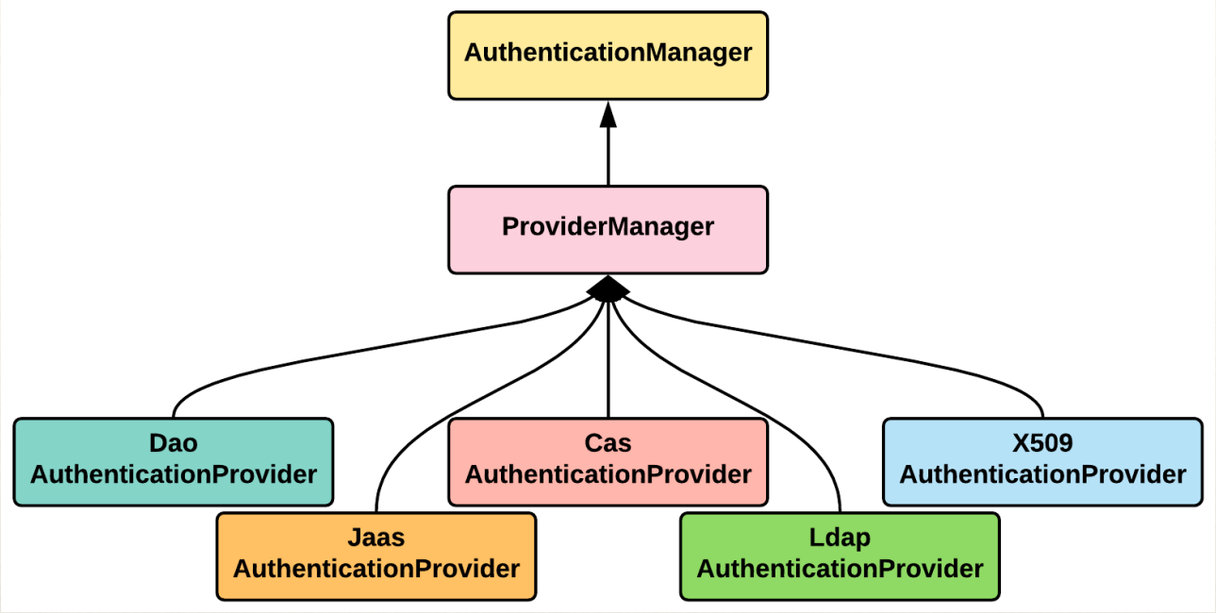
- Spring Security also has a number of helper classes, using which you can set up `AuthenticationManager`.
- One helper class is `AuthenticationManagerBuilder`.
- Using this class, it is easy to set up `UserDetailsService` against a database, in memory, in LDAP, and so on.
- You can make an `AuthenticationManager` global, so it will be accessible by your entire application.
- It will be available for method security and other `WebSecurityConfigurerAdapter` instances.
- `WebSecurityConfigurerAdapter` is a class that is extended by your Spring configuration file, making it quite easy to bring Spring Security into your Spring application.
- This is how you set up a global `AuthenticationManager` using the `@Autowired` annotation:

```
@Configuration
@EnableWebSecurity
public class SpringSecurityConfig extends WebSecurityConfigurerAdapter {

    @Autowired
    public void configureGlobalAuthenticationManager(AuthenticationManagerBuilder
auth) throws Exception {
        auth.inMemoryAuthentication()
                .withUser("admin").password("admin@password").roles("ROLE_ADMIN");
    }
}
```

Spring Security

AuthenticationProvider




```java
@Component
public class CustomAuthenticationProvider implements AuthenticationProvider {
    @Override
    public Authentication authenticate(Authentication authentication) throws AuthenticationException {
      String username = authentication.getName();
      String password = authentication.getCredentials().toString();
      if ("user".equals(username) && "password".equals(password)) {
        return new UsernamePasswordAuthenticationToken
          (username, password, Collections.emptyList());
      } else {
        throw new BadCredentialsException("Authentication failed");
      }
    }
    @Override
    public boolean supports(Class<?> aClass) {
      return aClass.equals(UsernamePasswordAuthenticationToken.class);
    }
}
```

Spring Security

Multiple AuthenticationProvider

- Spring Security allows you to declare multiple `AuthenticationProvider` in your application.
- They are executed according to the order in which they are declared in the configuration.
- For example, `CustomAuthenticationProvider` as an `AuthenticationProvider` (`Order 1`) and the existing `inMemoryAuthentication` as our second `AuthenticationProvider` (`Order 2`):
- Whenever the `authenticate` method executes without error, the controls return and thereafter configured `AuthenticationProvider`'s doesn't get executed.

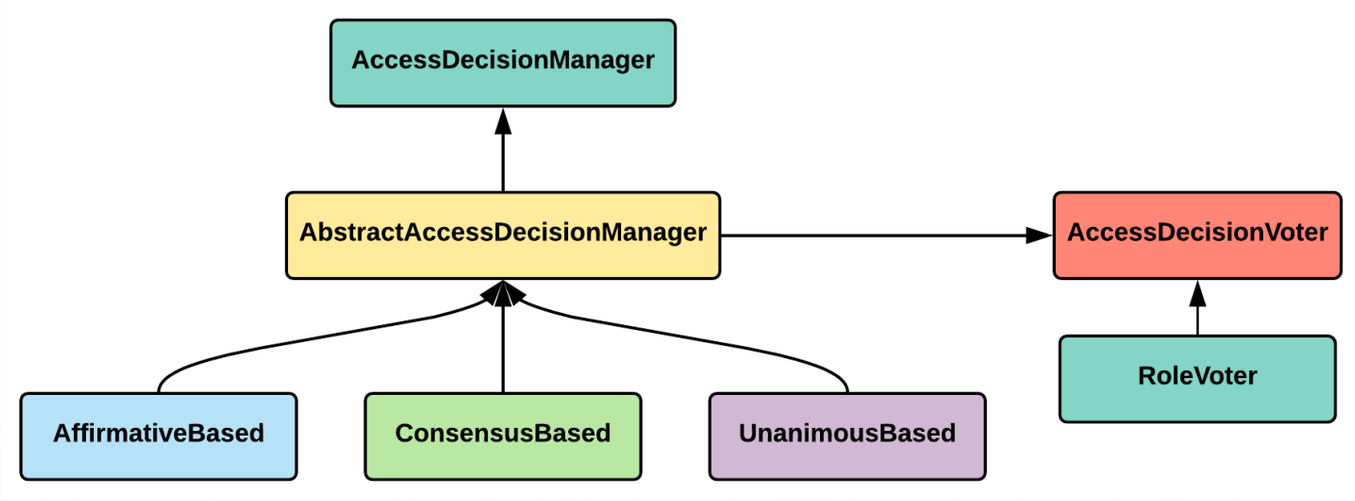
```
@EnableWebSecurity
@ComponentScan(basePackageClasses =
CustomAuthenticationProvider.class)
public class SpringSecurityConfig extends
WebSecurityConfigurerAdapter {

@Autowired
CustomAuthenticationProvider customAuthenticationProvider;

@Override
protected void configure(HttpSecurity http) throws Exception {
        http.httpBasic()
                .and()
                .authorizeRequests()
                .antMatchers("/**")
                .authenticated(); // Use Basic authentication
}
@Override
protected void configure(AuthenticationManagerBuilder auth) throws
Exception {
// Custom authentication provider - Order 1
auth.authenticationProvider(customAuthenticationProvider);
// Built-in authentication provider - Order 2
auth.inMemoryAuthentication()
                .withUser("admin")
                .password("{noop}admin@password")
//{noop} makes sure that the password encoder doesn't do anything
.roles("ADMIN") // Role of the user
.and()
                .withUser("user")
                .password("{noop}user@password")
                .credentialsExpired(true)
                .accountExpired(true)
                .accountLocked(true)
                .roles("USER");
    }
}
```

Spring Security

Authorization



- Spring Authorization also has a manager, `AccessDecisionManager`.
- Spring Security has three built-in implementations for this: `AffirmativeBased`, `ConsensusBased`, and `UnanimousBased`.
- `AccessDecisionManager` works by delegating to a chain of `AccessDecisionVoter`.
- In Spring Security, authorization to a secured resource is granted by invoking voters and then tallying the votes received.
- The three built-in implementations tally the votes received in different manners:
  - **AffirmativeBased**: If at least one voter votes, the user is given access to the secured resource
  - **ConsensusBased**: If a clear consensus is reached between the voters and their votes, then the user is given access to the secured resource
  - **UnanimousBased**: If all the voters vote, then the user is given access to the secured resource
- Spring Security provides two authorization approaches:
  - **Web URL**: Incoming URL (specific URL or regular expression)-based authorization
  - **Method**: Method signature based on which access is controlled

Spring Security

Use Case Exercises

- Spring MVC 5 + Spring Security 5 + XML + In Memory
- Spring MVC 5 + Spring Security 5 + Hibernate 5
- Spring MVC 5 + Spring Security 5 + Persistent Token

Spring Security

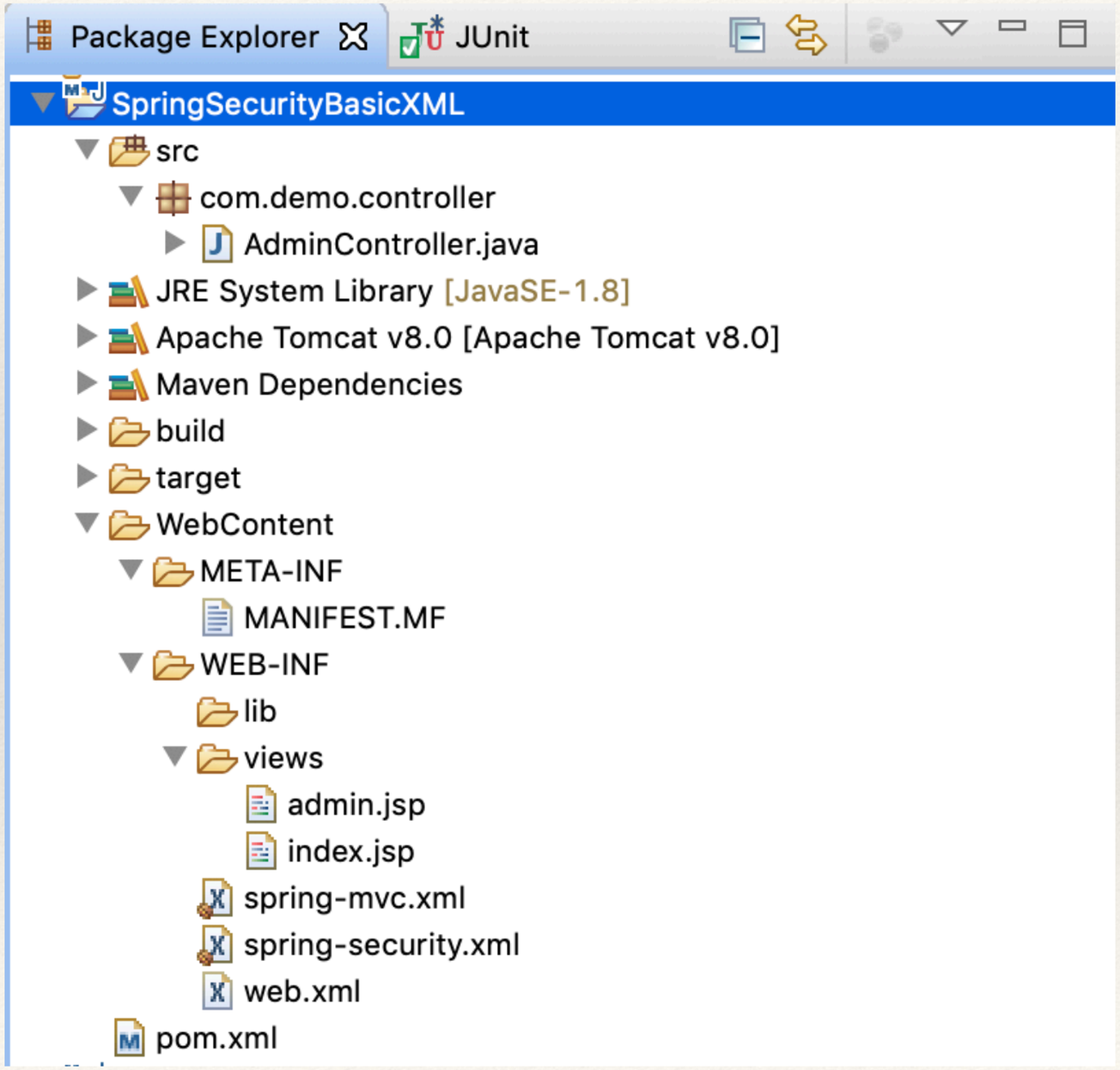
Use Case Exercise

# 1	Spring MVC 5 + Security + XML + In-memory
Project Name	**SpringSecurityBasicXML**
Environment	• Java 8. • Tomcat 8. • Spring framework 5.1.4.RELEASE. • Spring Security 5.1.4.RELEASE. • JSTL 1.2 • Eclipse IDE, STS Release (3.9.9)
About the Application	• It is a **Simple Spring Security** example to demonstrate XML configuration and uses **default login page** given by Spring Security • The sample **data of user & role** is configured within the **Security XML** file
Anything more to observe here?	• The **security filter** is declared within **web.xml** • There is a **spring security xml** file for **authentication & authorization** processing

Spring Security

Use Case Exercises

Project Structure - Spring MVC 5 + Security + XML + In-memory



Spring Security

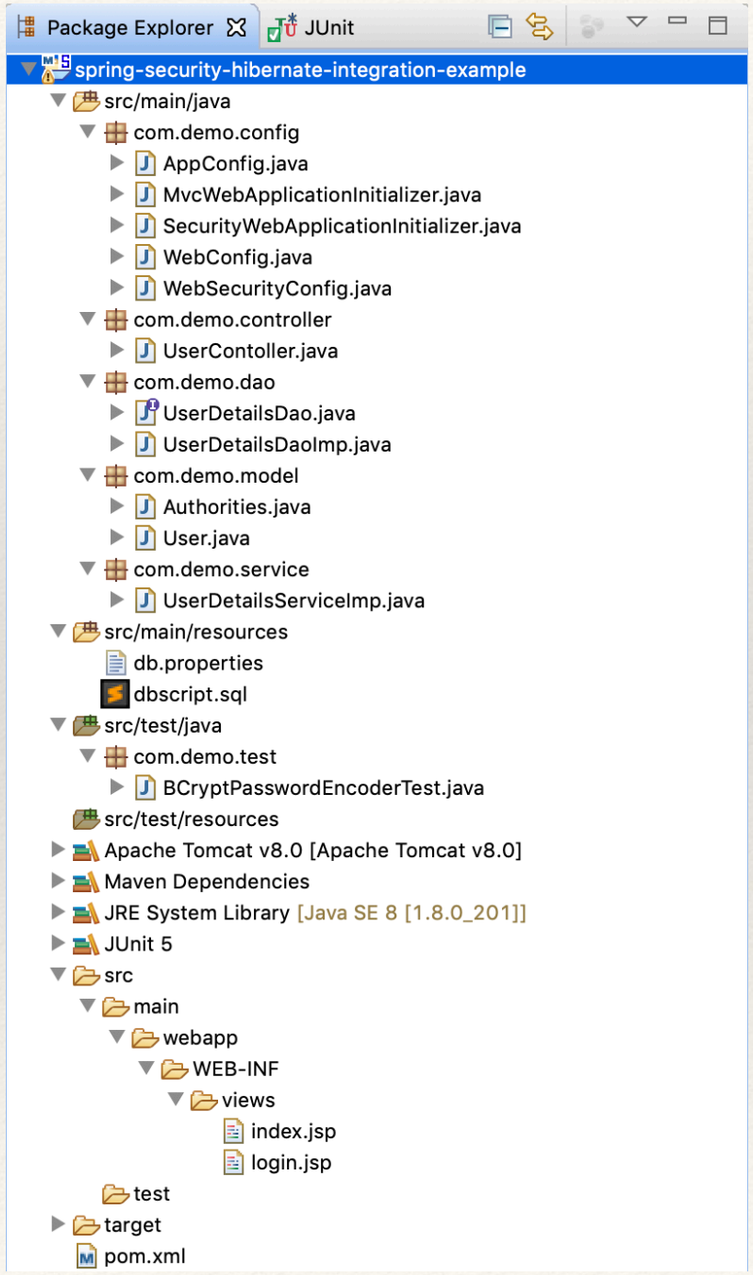
Use Case Exercise

# 2	Spring MVC 5 + Spring Security 5 + Hibernate 5
Project Name	**spring-security-hibernate-integration-example**
Environment	• Spring Security 5.0.0.RELEASE, Spring MVC 5.0.2.RELEASE • Spring ORM 5.0.2.RELEASE, Hibernate 5.2.12.Final • C3p0 0.9.5.2, Servlet API 3.1.0 • Common Pool 2.1.1, Java SE 8 • Maven 3.2 • Spring STS (3.9.9) • Tomcat 8.x • MySQL Server 5.7
About the Application	• How to create a **custom login form** in Spring MVC application with Spring Security • How to **integrate the Hibernate with Spring security** framework to load the user's authentication
DB Script	• File = src/main/resources/dbscript.sql • Encode the password before insertion
Anything more to observe here?	• How to use the **UserDetailsService interface** to load the **user's authentication** information from a database

Spring Security

Use Case Exercise

Project Structure - Spring MVC 5 + Spring Security 5 + Hibernate 5



Spring Security

Spring Security with Persistent Tokens

- Remember-me or persistence-login authentication allows websites to remember the identity of a logged-in user between sessions.
- Spring Security sends a cookie to browser, when user login to the application with remember-me option.
- This cookie is stored at browser side for a specific time period.
- Next time, when user logins to application, Spring security will check and validate the stored cookie and cause the automatic login if cookie is valid.
- Spring Security provides two approaches for implementing remember-me -
  - **Hash-Based Token Approach** - In this approach, username, expiration time, password and a private key are hashed and sent to browser as a token.
  - **Persistent Token Approach** - In this approach, a database or other persistent storage mechanism is used to store the generated tokens.
- Hash-Based token approach has a potential security issue because it contains the password in hash string.
- To enable remember-me authentication, you need to invoke the **rememberMe()** method of the **HttpSecurity** class and to store the generated tokens in database table invoke the **tokenRepository()** method with **PersistentTokenRepository** argument.

Spring Security

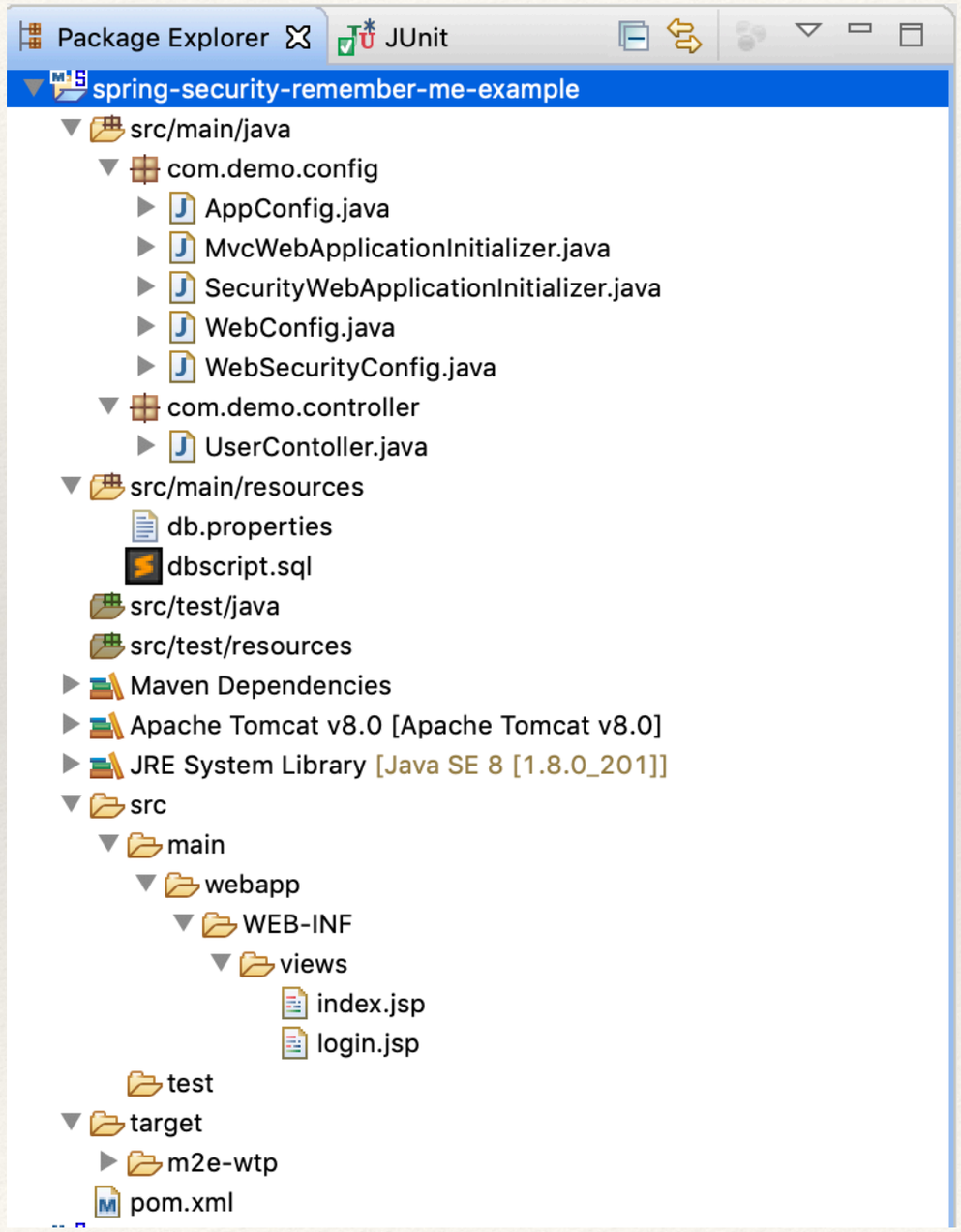
Use Case Exercise

# 3	Spring MVC 5 + Spring Security 5 + Persistent Token
Project Name	**spring-security-remember-me-example**
Environment	• Spring Security 5.0.0.RELEASE, Spring MVC 5.0.2.RELEASE • Spring JDBC 5.0.2.RELEASE, Servlet API 3.1.0 • Common Pool 2.1.1, Java SE 8 • Maven 3.2 • Spring STS (3.9.9) • Tomcat 8.x • MySQL Server 5.7
About the Application	• How to implement the **Remember-me authentication** in Spring MVC web application using the **Persistent Token Approach**.
DB Script	• File = src/main/resources/dbscript.sql • **Encode the password before insertion**
Anything more to observe here?	• Using **rememberMe() method from HttpSecurity** for remember-me authentication • Observe cookies storage in the Browser

Spring Security

Use Case Exercise

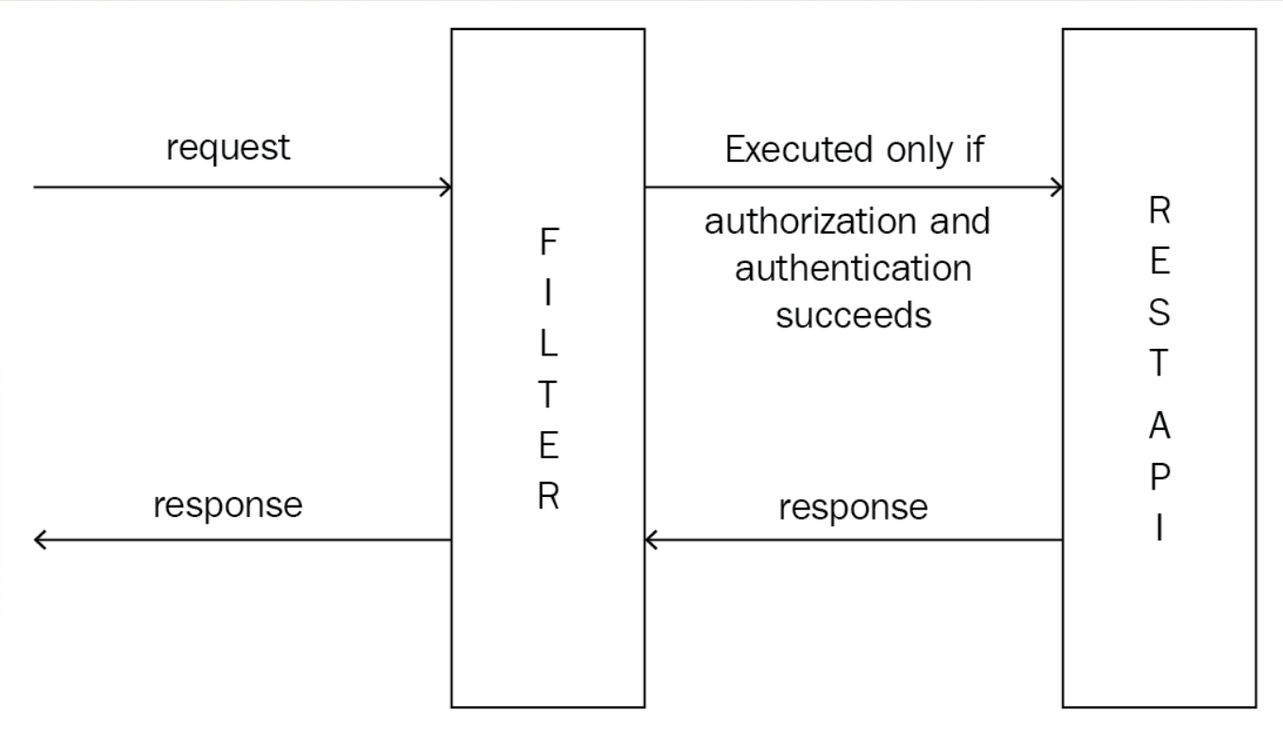
Project Structure - Spring MVC 5 + Spring Security 5 + Persistent Token



Spring Security

Securing REST API with Spring Security

- The popular option is to have **multiple roles** associated with the **API**.
- We can have permissions for different API actions that are associated with the roles;
- For example:
  - The user role can perform all actions on their todos—`Read`, `Update`, `Delete`, and `Create`.
  - The support role can only view the todos—only `Read` is allowed.
- **REST APIs** are implemented in the **Java** world using **servlets and controllers**.
- We want to ensure that the authorization and authentication are performed before the servlets and/or controllers are executed.
- Filters provide this feature. We can execute filters before the API is executed.



Spring Security

Securing REST API with Spring Security

- Adds mandatory authentication for URLs
- Adds login form
- Handles login error
- Create a user and sets a default password

Spring Security

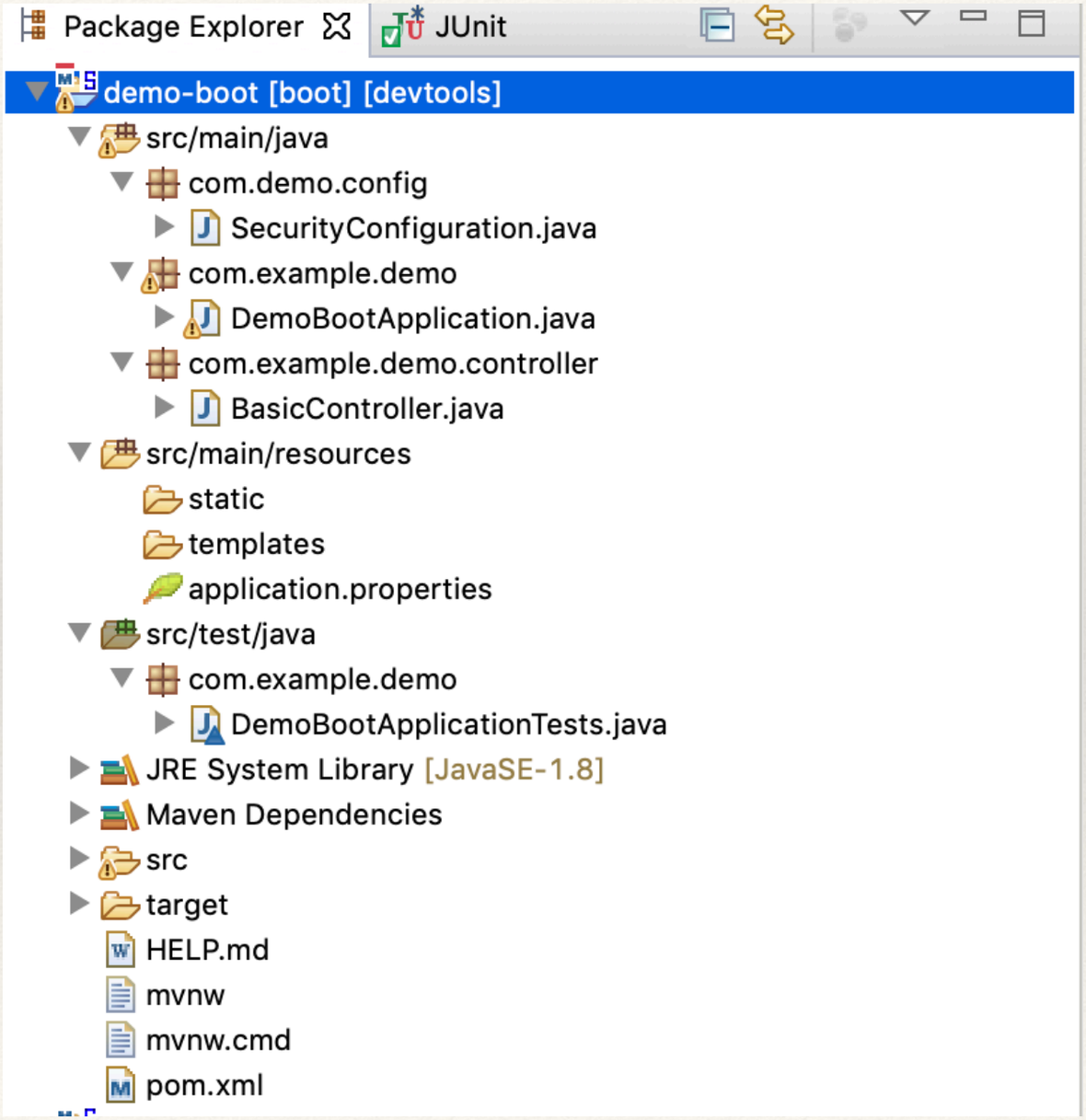
Use Case Exercises

# 4	Spring Boot + Security
Project Name	**demo-boot-withoutsecurity** & **demo-boot**
Environment	• Spring Boot Starter 2.2.0.RELEASE, Spring Boot Starter Web 2.2.0.RELEASE • Spring Boot Starter Security 2.2.0.RELEASE • Java SE 8 • Maven 3.2 • Spring STS (3.9.9) • Tomcat 8.x
About the Application	• Simple Spring Boot Application with Spring Security • Spring Boot Security handles basic security configuration internally
Anything more to observe here?	• It provides default credentials from the underlying security framework

Spring Security

Use Case Exercises

Project Structure - Spring Boot + Spring Security



Spring Security

Use Case Exercises

# 5	Spring Boot + Spring Security + REST API + MySQL + InMemory Users
Project Name	**boot-security-loginwith-users**
Environment	• Spring Boot Starter 2.2.0.RELEASE, Spring Boot Starter Web 2.2.0.RELEASE • Spring Boot Starter Security 2.2.0.RELEASE • Java SE 8 • Maven 3.2 • Spring STS (3.9.9) • MySQL 8.x / Oracle • Tomcat 8.x

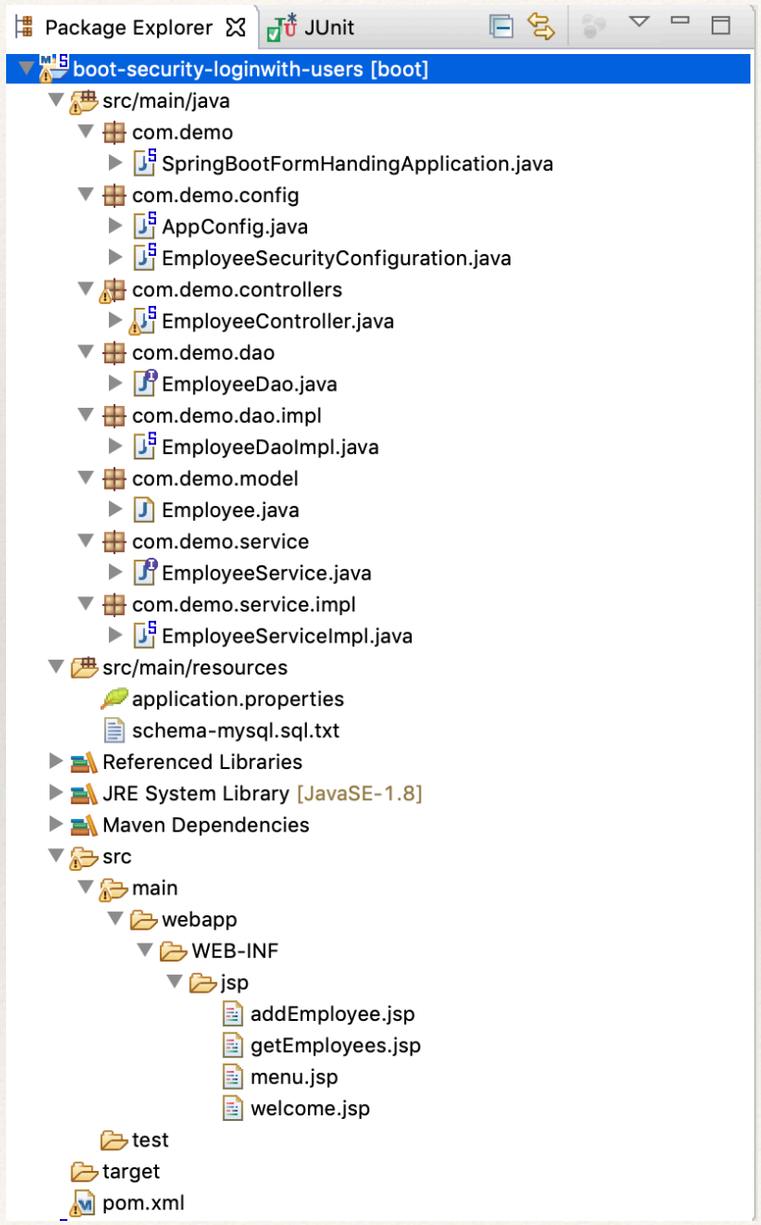
About the Application	• Spring Boot REST API for Employees Create & Retrieve operations
	• It demonstrates REST API Security where:
	  • User "employee" can access only getEmployees API
	  • User "admin" can access API to add a new employee
	• **The security login page is by default given by Spring Boot Security**
	• It connects to MySQL/Oracle

Anything more to observe here?	• Both the users must be able to fetch employees • **After logout, the login redirection is not working**, so after logout you need to change the URL to: **http://localhost:8080/welcome**

Spring Security

Use Case Exercises

Project Structure - Spring Boot + Spring Security + REST API + MySQL + InMemory Users



Spring Security

Use Case Exercises

# 6	Spring Boot + Spring Security + REST API + MySQL + InMemory Users + Custom Login Form
Project Name	**boot-security-custom-loginform**
Environment	• Spring Boot Starter 2.2.0.RELEASE, Spring Boot Starter Web 2.2.0.RELEASE • Spring Boot Starter Security 2.2.0.RELEASE • Java SE 8 • Maven 3.2 • Spring STS (3.9.9) • MySQL 8.x / Oracle • Tomcat 8.x

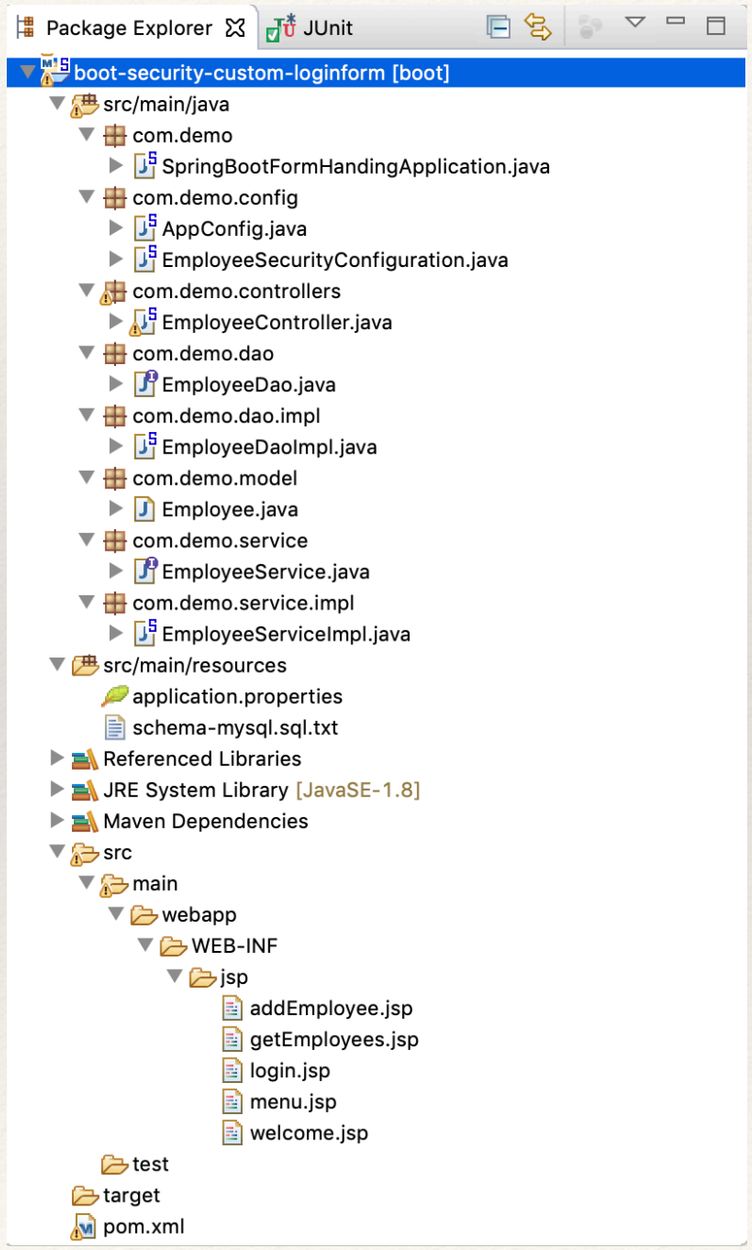
About the Application	• Spring Boot REST API for Employees Create & Retrieve operations
	• It demonstrates REST API Security where:
	  • User "employee" can access only getEmployees API
	  • User "admin" can access API to add a new employee
	• **The security login page is a custom login page with a /login API in the REST Controller**
	• It connects to MySQL/Oracle
	• User credentials are provided in-memory within SecurityConfig class

Anything more to observe here?	• Both the users must be able to fetch employees • **After logout, the login redirection is not working**, so after logout you need to change the URL to: **http://localhost:8080/welcome**

Spring Security

Use Case Exercises

Project Structure - Spring Boot + Spring Security + REST API + MySQL + InMemory Users + Custom Login Form



Spring Security

Use Case Exercises

# 7	Spring Boot + Spring Security + REST API + MySQL + DB Authentication + Custom Login Form
Project Name	**boot-security-custom-form-jdbc-auth**
Environment	• Spring Boot Starter 2.2.0.RELEASE, Spring Boot Starter Web 2.2.0.RELEASE • Spring Boot Starter Security 2.2.0.RELEASE • **Spring Boot Starter JDBC 2.2.0.RELEASE** • Java SE 8 • Maven 3.2 • Spring STS (3.9.9)

About the Application	• Spring Boot REST API for Employees Create & Retrieve operations
	• It demonstrates REST API Security where:
	  • User "employee" can access only getEmployees API
	  • User "admin" can access API to add a new employee
	• **The security login mechanism authenticates & authorizes from DB**
	• It connects to MySQL/Oracle for Employee API
	• User credentials are provided in-memory within SecurityConfig class

DB Scripts	schema_mysql.sql.txt
Anything more to observe here?	• Both the users must be able to fetch employees • **After logout, the login redirection is not working**, so after logout you need to change the URL to: **http://localhost:8080/welcome**

Spring Security

Use Case Exercises

# 8	Spring Boot + Spring Security + REST API + MySQL + DB Authentication + Custom Login Form + JdbcUserDetails
Project Name	**Boot-security-jdbc-auth-create-users**
Environment	• Spring Boot Starter 2.2.0.RELEASE, Spring Boot Starter Web 2.2.0.RELEASE • Spring Boot Starter Security 2.2.0.RELEASE • **Spring Boot Starter JDBC 2.2.0.RELEASE** • Java SE 8 • Maven 3.2 • Spring STS (3.9.9) • MySQL 8.x / Oracle • Tomcat 8.x

About the Application	• Spring Boot REST API for Employees Create & Retrieve operations
	• It demonstrates REST API Security where:
	  • User "employee" can access only getEmployees API
	  • User "admin" can access API to add a new employee
	  • **New user creation before authentication**
	• The security login mechanism authenticates & authorizes from DB
	• It connects to MySQL/Oracle for Employee API

DB Scripts	schema_mysql.sql.txt
Anything more to observe here?	• **After logout, the login redirection is not working**, so after logout you need to change the URL to: **http://localhost:8080/welcome**

Spring Security

Use Case Exercises

# 9	Spring Boot + Spring Security + REST API + JdbcUserDetails + Password Encode
Project Name	**Boot-security-jdbc-create-users-pwencode**
Environment	• Spring Boot Starter 2.2.0.RELEASE, Spring Boot Starter Web 2.2.0.RELEASE • Spring Boot Starter Security 2.2.0.RELEASE • Spring Boot Starter JDBC 2.2.0.RELEASE • Java SE 8 • Maven 3.2 • Spring STS (3.9.9) • MySQL 8.x / Oracle • Tomcat 8.x

About the Application	• Spring Boot REST API for Employees Create & Retrieve operations
	• It demonstrates REST API Security where:
	  • User "employee" can access only getEmployees API
	  • User "admin" can access API to add a new employee
	  • New user creation before authentication
	• The security login mechanism authenticates & authorizes from DB
	• It connects to MySQL/Oracle for Employee API
	• User credentials are provided in-memory within SecurityConfig class
	• **Encoded passwords using BCrypt**

DB Scripts	schema_mysql.sql.txt
Anything more to observe here?	• **After logout, the login redirection is not working**, so after logout you need to change the URL to: **http://localhost:8080/welcome**

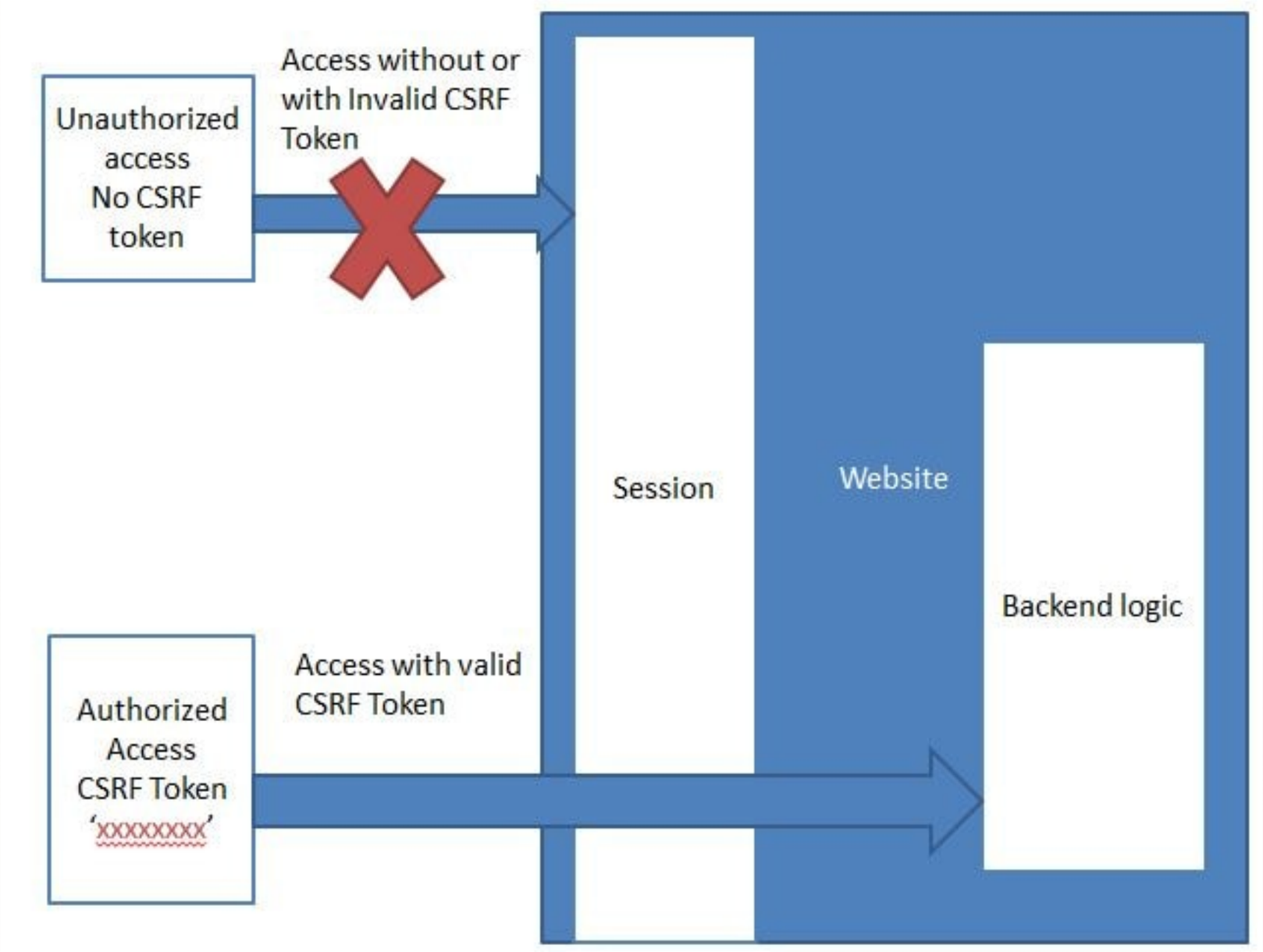
Spring Security

Spring Boot Security - Enabling CSRF Protection

- CSRF stands for **Cross-Site Request Forgery**.
- It is an attack that forces an end user to **execute unwanted actions** on a web application in which they are currently authenticated.
- CSRF attacks specifically target **state-changing requests**, not theft of data, since the attacker has no way to see the response to the forged request.

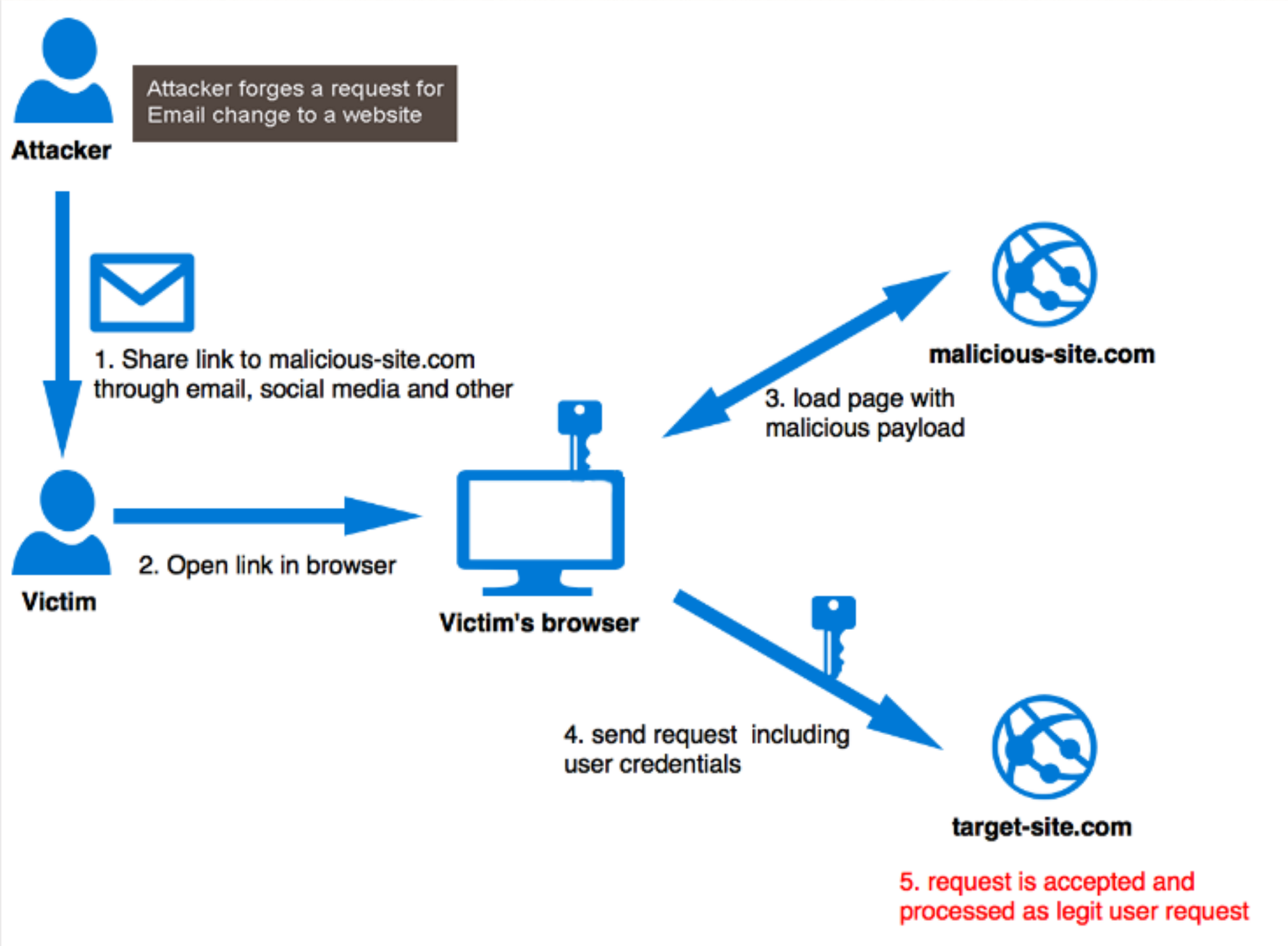
Spring Security

Spring Boot Security - Enabling CSRF Protection



Spring Security

Spring Boot Security - Enabling CSRF Protection



Spring Security

Spring Boot Security - Enabling CSRF Protection

- CSRF stands for **Cross-Site Request Forgery**.
- It is an attack that forces an end user to **execute unwanted actions** on a web application in which they are currently authenticated.
- CSRF attacks specifically target **state-changing requests**, not theft of data, since the attacker has no way to see the response to the forged request.

```
<dependency>
   <groupId>org.springframework.security</groupId>
   <artifactId>spring-security-taglibs</artifactId>
</dependency>
```

- **Config.java**

```
//http.csrf().disable();
```

- **Use Spring Form Tag Library - <form:form>**

```
<%@ taglib prefix="sec" uri="http://www.springframework.org/security/tags" %>
<sec:csrfInput />
```

Spring Security

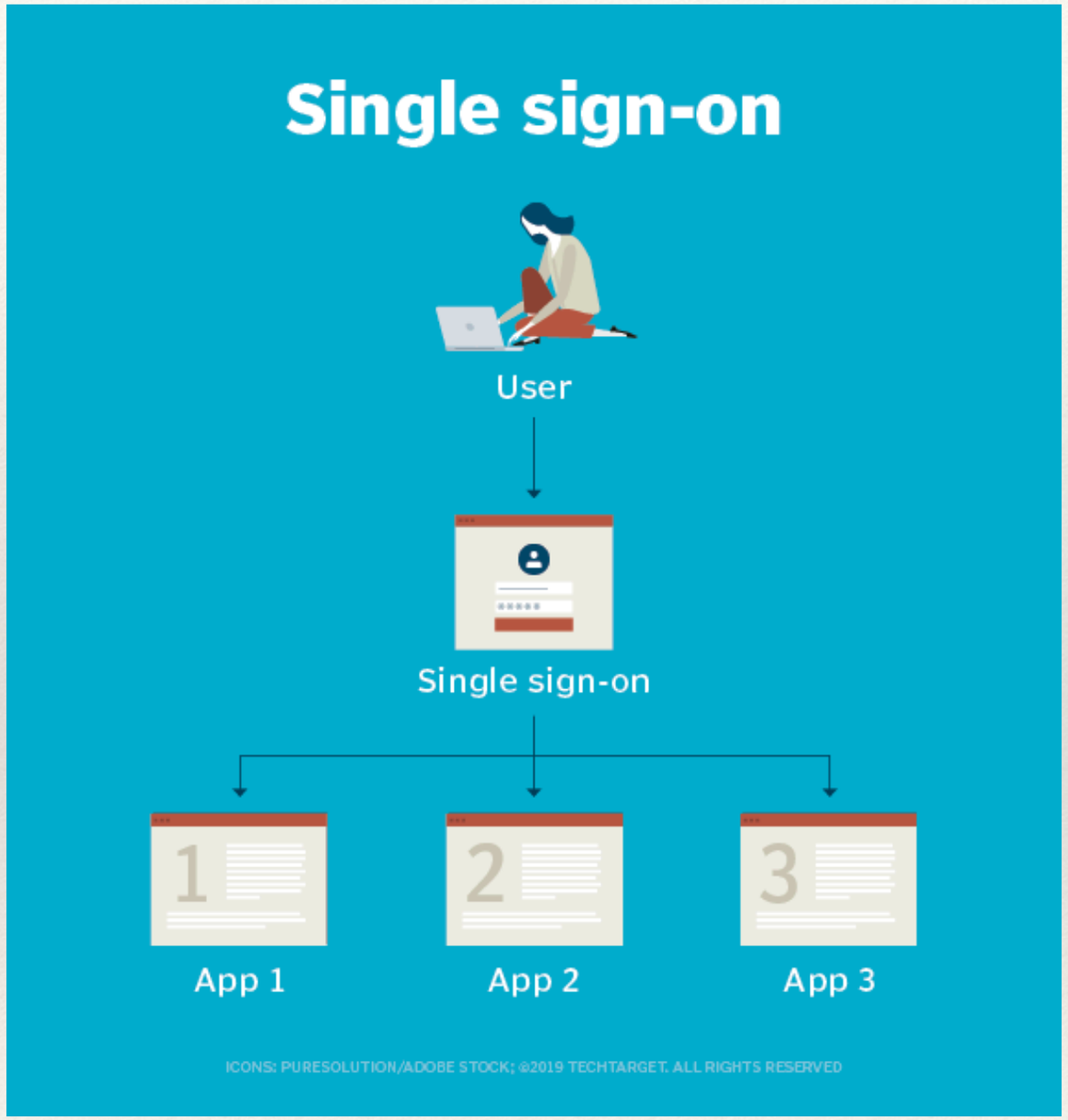
Understanding SSO

*"**Single sign-on (SSO)** is a session and user authentication service that permits a user to use one set of login credentials (e.g., name and password) to access multiple applications."*

- In a basic web SSO service, an Agent module on the application server retrieves the **specific authentication credentials** for an individual user from a dedicated **SSO policy server**, while authenticating the user against a user repository
- The service authenticates the end user for **all the applications** the user has been given rights to and eliminates future password prompts for individual applications during the same session.

Spring Security

Understanding SSO



Spring Security

Understanding SSO

Types of SSO configurations	
SAML	• SAML is an XMLstandard that facilitates the exchange of user authentication and authorization data across secure domains. • SAML-based SSO services involve communications between the user, an identity provider that maintains a user directory, and a service provider.
Kerberos	• Once the user credentials are provided, a ticket-granting ticket (**TGT**) is issued. • The **TGT** fetches service tickets for other applications the user wishes to access, without asking the user to re-enter credentials
Smart-card based SSO	• Smart-card-based SSO will ask an end user to use a card holding the sign-in credentials. • Once first used, a user will not have to re-enter usernames or passwords. SSO smart cards will store either certificates or passwords.

Spring Security

Understanding SSO

- There are security risks for SSO as well
- SSO implementation can be coupled with Identity Governance
- 2FA or MFA is a common suction

Spring Security

Social SSO

Advantages of SSO include:

- Allows users to remember and **manage less passwords** and usernames for each application.
- Streamlines the **process of signing** on and using applications—no need to re-enter passwords.
- Lessens the chance of **phishing**
- Less complaints or trouble about passwords for IT helpdesks.

Disadvantages of SSO include:

- It does not address certain levels of security each application sign-on may need.
- If availability is **lost**, then users are locked out of the multiple systems connected to the SSO.
- If an unauthorized user gains access, then **access could gain access to more than one application.**

Spring Security

SSO Vendors

	Average Rating	Reviews	Words/Review	Views	Comparisons
1. Okta	8.6	7	254	22,109	17,958
2. Keeper	9.7	10	196	8,432	3,858
3. CA SSO	8.0	14	653	3,965	2,033

Spring Security

SSO Vendors

4. LastPass Enterprise	7.4 Average Rating	9 Reviews	772 Words/Review	7,315 Views	4,982 Comparisons
5. FortiAuthenticator	8.8 Average Rating	5 Reviews	392 Words/Review	5,558 Views	3,521 Comparisons
6. Auth0	9.0 Average Rating	1 Reviews	367 Words/Review	7,943 Views	6,233 Comparisons

Spring Security

SSO Vendors

7. OneLogin	8.0 Average Rating	1 Reviews	842 Words/Review	6,386 Views	5,226 Comparisons
8. Tools4ever HelloID	9.8 Average Rating	4 Reviews	526 Words/Review	556 Views	298 Comparisons

Spring Security

SSO - The Best Solutions

- https://www.business.com/categories/best-single-sign-on-solutions/

Spring Security

Security Assertion Markup Language (SAML)

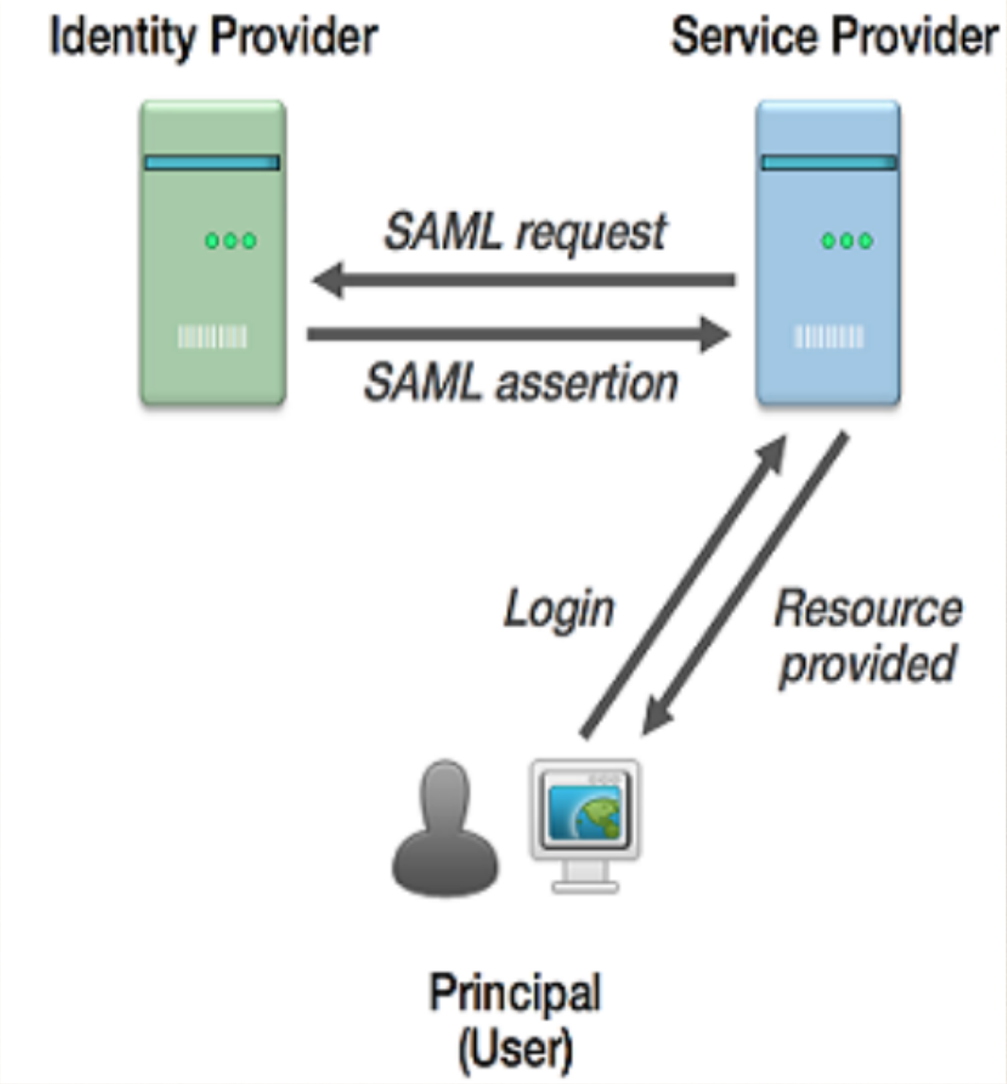
- SAML is developed by the Security Services Technical Committee of OASIS, is an XML-based framework for **communicating user authentication, entitlement** and attribute information.
- SAML allows business entities to make **assertions** regarding the identity, attributes, and entitlements of a **subject** (an entity that is often a human user) **to other entities**, such as a **partner company or another enterprise.**
- SAML is also
  - A set of XML-based protocol messages
  - A set of protocol message bindings
  - A set of profiles (utilizing all of the above)

Spring Security

Security Assertion Markup Language (SAML)

- **Identity Provider (IdP)** is a system that creates, maintains, and manages identity information for principals (users, services, or systems), and provides principal authentication to other service providers (applications) within a federation or distributed network.

- **Service Provider (SP)** is any system that provides services, typically the services for which users seek authentication, including web or enterprise applications.



Spring Security

SAML support for Spring

- Spring Security has a top-level project named **Spring Security SAML**.
- It is considered an extension providing Spring applications to integrate with a variety of authentication and federation mechanisms that supports SAML 2.0.
- This extension also supports multiple SAML 2.0, profiles as well as IdP and SP initiated SSO.
- There are a number of SAML 2.0 compliant products (IdP mode), such as **Okta, Ping Federate, and ADFS**, that can be integrated into your application quite easily using this Spring Security extension.

Spring Security

Use Case Exercises

# 11	Spring Boot + SAML + OKTA
Project Name	**spring-boot-saml-demo**
Environment	• Spring Boot Starter 2.0.1.RELEASE, Spring Boot Starter Web 2.0.1.RELEASE • Spring Boot Starter Security 2.0.1.RELEASE • Spring Security SAML 2 Core 1.0.3 • Spring Security SAML DSL Core 1.0.5 • ***http://developer.okta.com*** • Java SE 8 • Maven 3.2 • Spring STS (3.9.9) • Tomcat 8.x

About the Application	• Spring Boot REST API
	• It demonstrates API Security where:
	  • Security using **OKTA**
	  • Requires **keystone file**
	• The application is enabled with **8443 for https**

Anything more to observe here?	• Uses **application.yml**

Spring Security

OAuth 2.0

- OAuth is an **open standard/specification** for achieving **authorization**. The specification works by validating access tokens, and then authorizes devices, APIs, servers, and so on.
- Two versions—namely OAuth 1.0 (**https://tools.ietf.org/html/rfc5849**) and OAuth 2.0 (**https://tools.ietf.org/html/rfc6749**)—exist. These versions are **not compatible** with each other and cannot work together.
- SAML, released in 2005, is a good fit for the web browser (still). But with modern web and native applications (mobile devices), SAML required a serious overhaul, and that's when OAuth came in.
- **SPAs do AJAX/XHR calls to the APIs** that are exposed on the server and does many other operations on the client (browser).

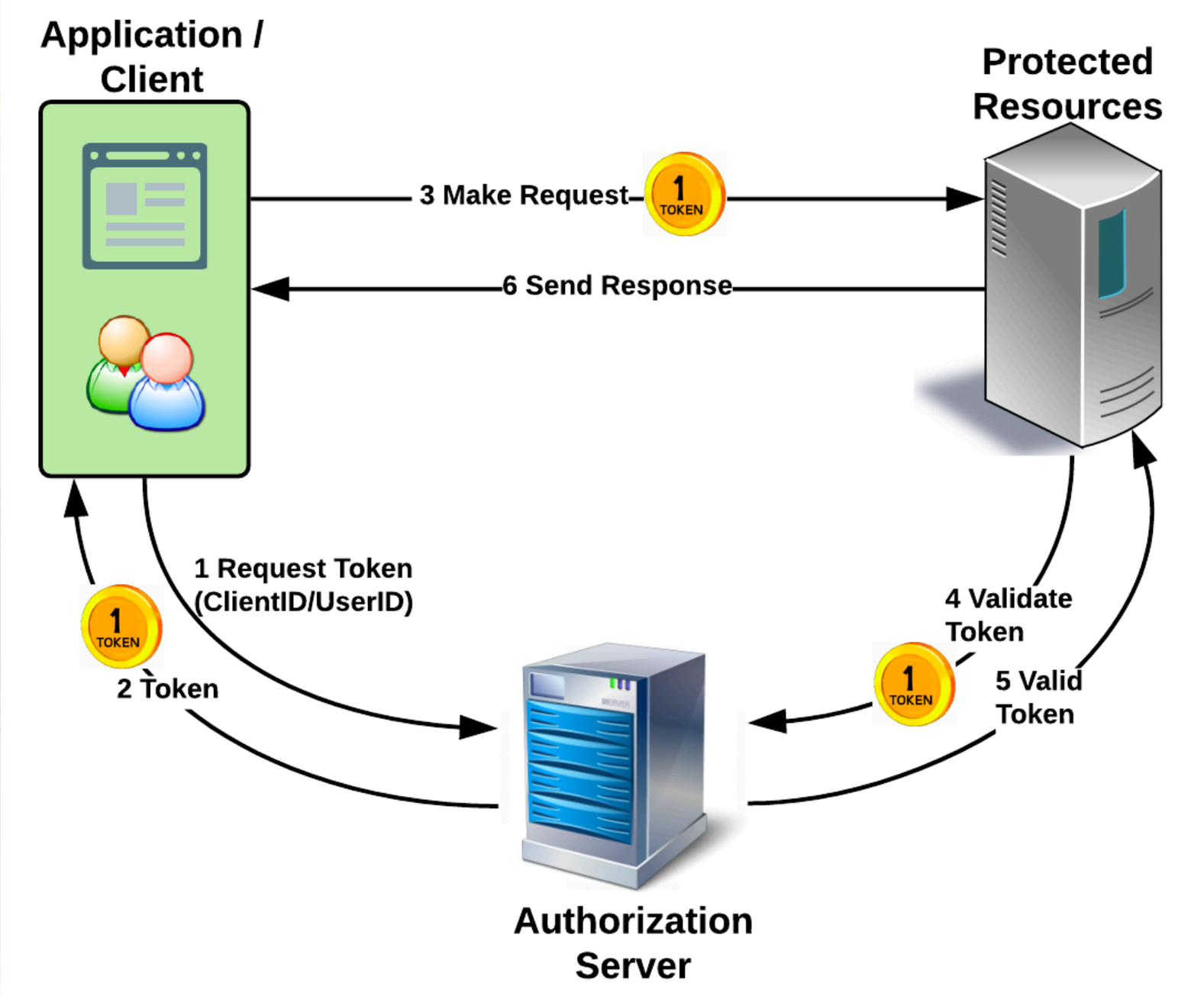
Spring Security

OAuth 2.0

- OAuth also enables you, as a developer, to gain access to **minimal user data** without having to give away a user's password.
- It is mainly for accessing the APIs (REST) exposed by an application, and is done by **delegating the authorization function**.
- OAuth supports a variety of application types and decouples authentication from authorization.
- **In simple terms, this is how OAuth works:**
  - The app that wants to access resources requests the user to grant authorization.
  - If the user authorizes it, the app is given proof for this agreement.
  - Using this proof, the app goes to the actual server with the APIs and gets a token.
  - Using this token, the app can now ask for resources (APIs) to which the user has given access, while giving the proof.

Spring Security

OAuth 2.0



Spring Security

OAuth & OIDC

- OAuth was tweaked in such a way by using an **access token**, apps can get user information in the form of an API.
- **Facebook Connect** (an SSO application that allows users to interact with other web applications using Facebook credentials) used this as a mechanism to expose an endpoint **(http(s)://<domain>/me)** that would return minimal user information.
- This was never clearly there in OAuth specification, and this provoked Open ID Connect (OIDC), which combined the **best parts of OAuth2, Facebook Connect, and SAML 2.0**. OIDC brought in a new ID token (**id_token**), and also a **UserInfo endpoint** that will provide minimal user attributes.
- Many of the complexities that SAML had, and many of the shortcomings of OAuth2 were addressed by OIDC.

Spring Security

Use Case Exercises

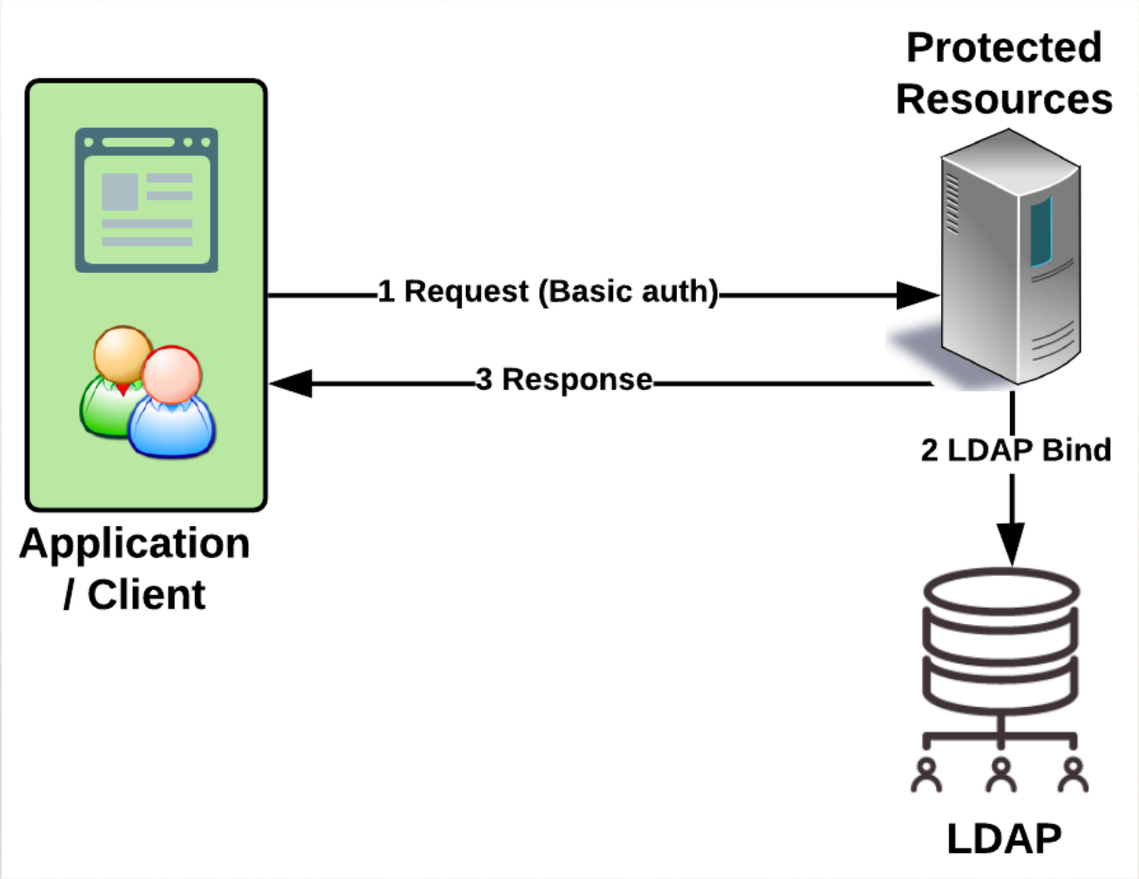
# 11	Spring Boot + OAuth2 using Google API
Project Name	**spring-boot-oauth-google-demo**
Environment	• Spring Boot Starter 2.2.0.RELEASE, Spring Boot Starter Web 2.2.0RELEASE • Spring Boot Starter Security 2.2.0.RELEASE • Spring Security OAuth2 Jose • Spring Security OAuth 2 Client • ***http://console.developer.google.com*** • Java SE 8 • Maven 3.2 • Spring STS (3.9.9) • Tomcat 8.x

About the Application	• Spring Boot REST API
	• It demonstrates API Security where:
	  • Security using **OAuth2 with Google API**
	  • Requires **Client ID & Client secret**

Anything more to observe here?	• Uses **application.properties and populates default web page with OAuth providers**

Spring Security

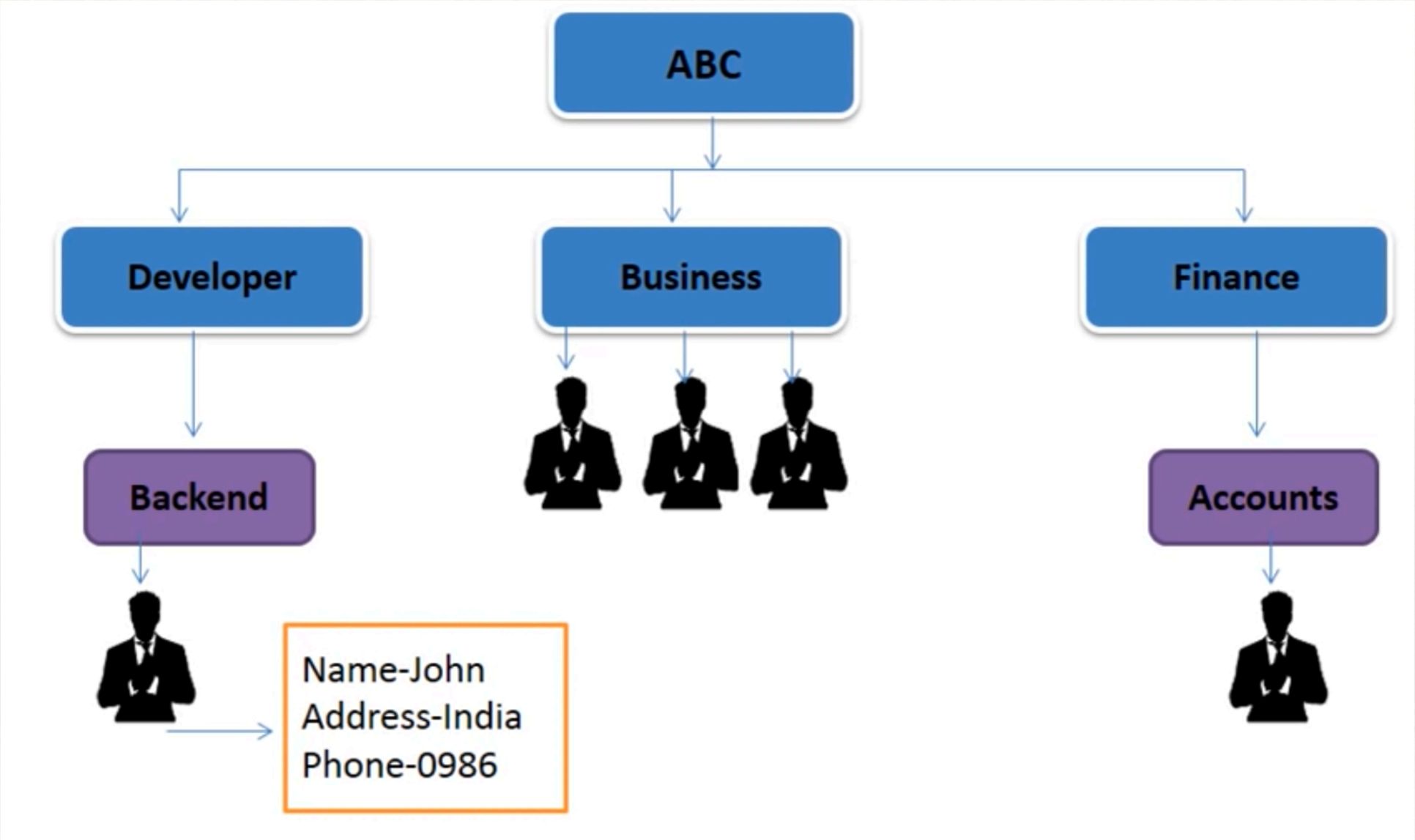
Lightweight Directory Access Protocol



"Lightweight Directory Access Protocol (LDAP) is a directory service protocol that allows for connecting, searching, and modifying internet directories. Unfortunately, LDAP doesn't support reactive bindings; this means that reactive programming is not possible (similar to JDBC) with it."

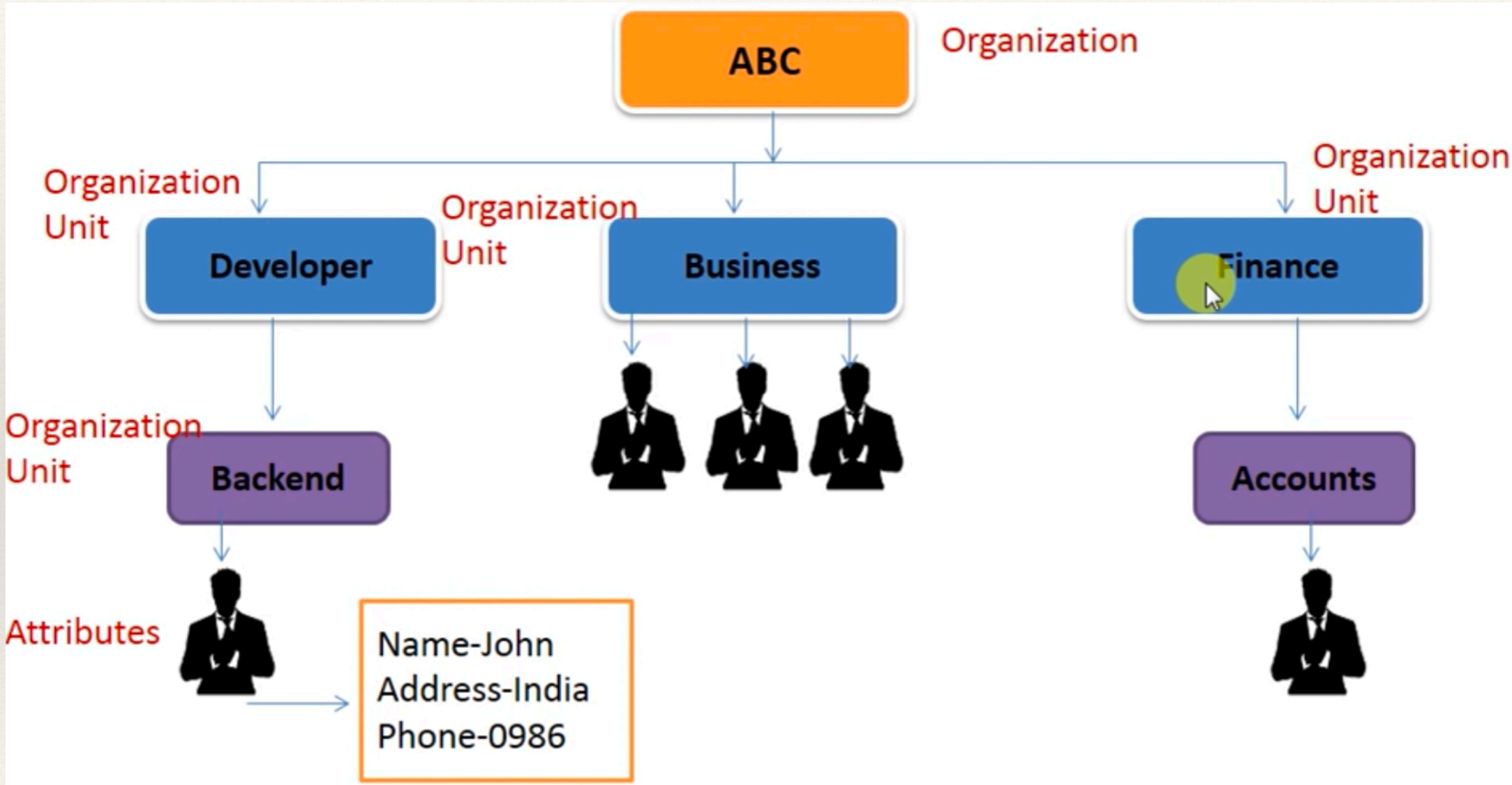
Spring Security

Lightweight Directory Access Protocol



Spring Security

Lightweight Directory Access Protocol



Spring Security

Lightweight Directory Access Protocol

cn – common name

sn – sur name

dn – distinguish name

o – organization name

ou – organization unit

User – inetOrgPerson

User – groupsOfUniqueName

Object

Spring Security

Use Case Exercises

# 12	Spring Boot + LDAP Authentication
Project Name	**spring-boot-ldap-demo**
Environment	• Spring Boot Starter 2.0.1.RELEASE, Spring Boot Starter Web 2.0.1.RELEASE • Spring Boot Starter Security 2.0.1.RELEASE • Spring LDAP Core • Spring Security LDAP • Java SE 8 • Maven 3.2 • Spring STS (3.9.9) • Tomcat 8.x

About the Application	• Spring Boot REST API
	• It demonstrates API Security where:
	  • Security using **LDAP Authentication**
	  • Requires **.ldif file**

Anything more to observe here?	• Starts LDAP server on 8389

Spring Security

JWT

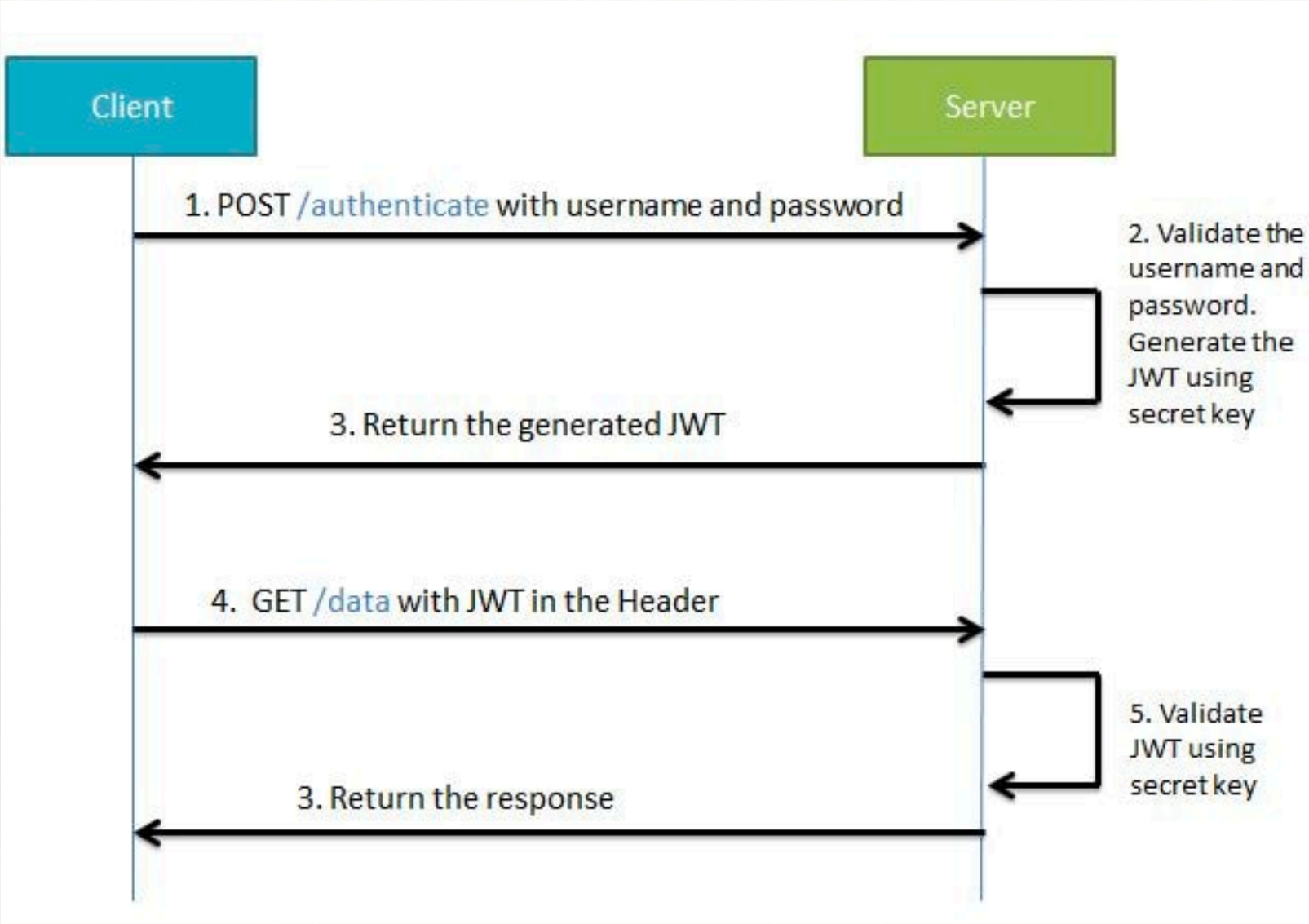
- **JSON Web Token (JWT)** is an open standard (RFC 7519) that defines a compact and self-contained way for securely transmitting information between parties as a JSON object. JWTs can be signed using a secret (with the HMAC algorithm) or a public/private key pair using RSA or ECDSA.

- **Authorization:** This is the most common scenario for using JWT. Once the user is logged in, each subsequent request will include the JWT, allowing the user to access routes, services, and resources that are permitted with that token. Single Sign On is a feature that widely uses JWT nowadays, because of its small overhead and its ability to be easily used across different domains.

- **Information Exchange:** JSON Web Tokens are a good way of securely transmitting information between parties. Because JWTs can be signed—for example, using public/private key pairs—you can be sure the senders are who they say they are. Additionally, as the signature is calculated using the header and the payload, you can also verify that the content hasn't been tampered with.

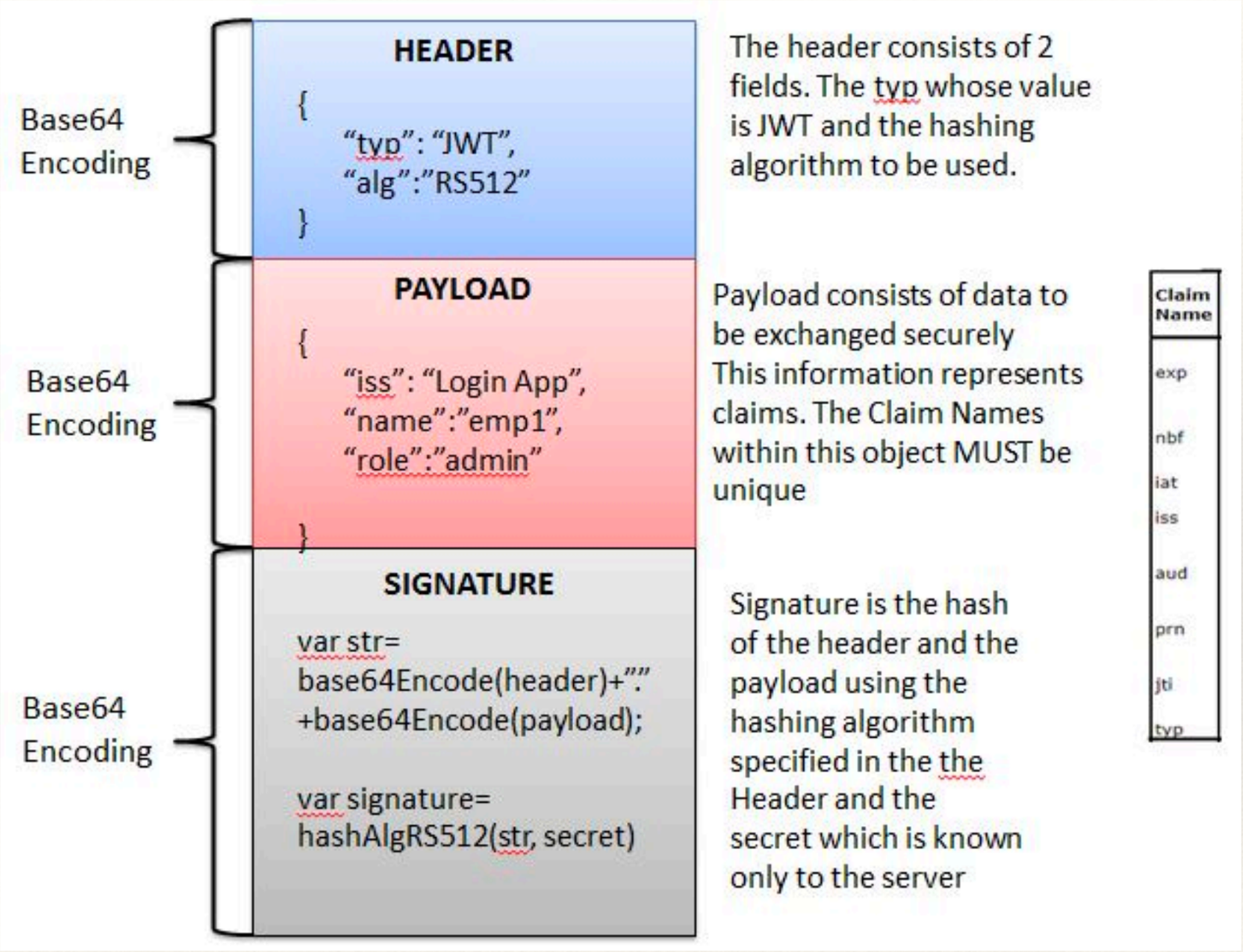
Spring Security

JWT



Spring Security

JWT



http://jwtbuilder.jamiekurtz.com/

Spring Security

Use Case Exercises

# 13	Spring Boot + JWT
Project Name	**spring-boot-jwt**
Environment	• Spring Boot Starter 2.1.1.RELEASE, Spring Boot Starter Web 2.1.1.RELEASE • Spring Boot Starter Security 2.1.1.RELEASE • JJWT 0.9.1 • Java SE 8 • Maven 3.2 • Spring STS (3.9.9) • Tomcat 8.x

About the Application	• Spring Boot REST API
	• It demonstrates API Security where:
	  • Security using **JSON Web Token**
	  • An **additional controller** to request for JWT token
	  • Token must be part of **HEADER** for each API request

Anything more to observe here?	• **Filter** checks for HEADER