Spring Security

prepared by- Vijay Kulkarni, Java Spring Trainer

Spring Security

- Spring Security, which is a powerful and **customizable authentication** and **authorization** framework.
- It is considered the de facto standard for securing Spring-based applications. e.g. if back end is based on Java and Spring, it makes sense to use Spring Security with JWT for authentication/authorization and configure it for stateless communication.

Terminology

- **Authentication** refers to the process of verifying the identity of a user, based on provided credentials. A common example is entering a username and a password when you log in to a website. You can think of it as an answer to the question *Who are you?*.
- **Authorization** refers to the process of determining if a user has proper permission to perform a particular action or read particular data, assuming that the user is successfully authenticated. You can think of it as an answer to the question *Can a user do/read this?*.
- **Principle** refers to the currently authenticated user.
- **Granted authority** refers to the permission of the authenticated user.
- **Role** refers to a group of permissions of the authenticated user.

Java Security Overview

- Java has a **comprehensive security model** designed into its architecture, built into **Java Virtual Machine** and hence preferred for **networks**.
- The **safety features** that **Java security** includes are large set of APIs, tools, and implementations of commonly-used security algorithms, mechanisms, and protocols.
- The **Java security APIs** span a wide range of areas, viz.. cryptography, public key infrastructure, secure communication, authentication, and access control.
- **Java security model** is focused on protecting users from hostile programs downloaded from **untrusted sources across a network**.
- Java provides a customizable "**sandbox**" in which Java programs run, mandatorily only inside its sandbox.

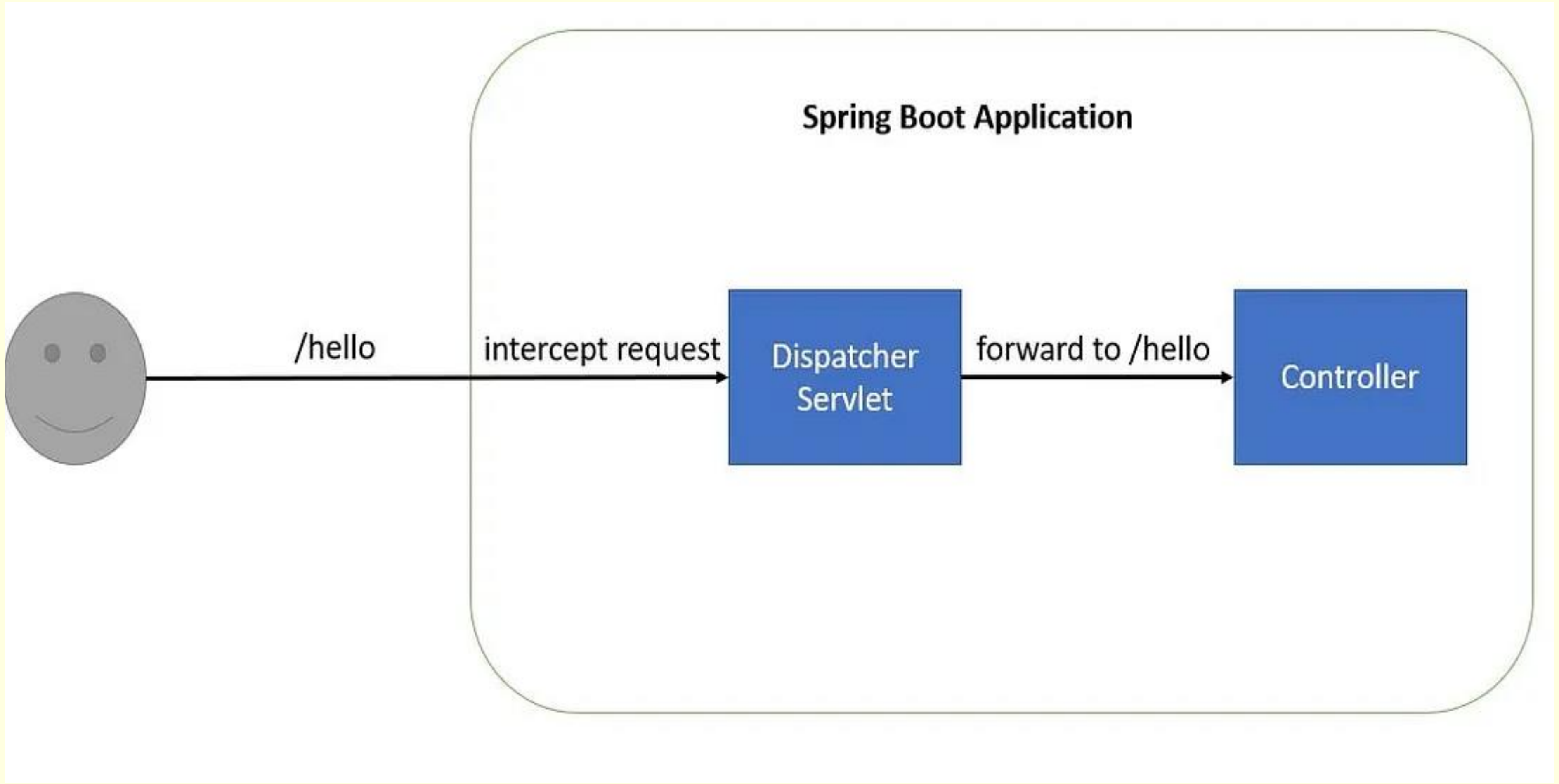
contd..

- The fundamental components responsible for Java's sandbox are:
  - ✓ Safety features built into the Java virtual machine and the language class loader architecture
  - ✓ The class file verifier
  - ✓ The security manager and the Java API
- Several built-in security mechanisms are operating as Java virtual machine bytecodes. The mechanisms are:
  - ✓ Type–safe reference casting
  - ✓ Structured memory access (no pointer arithmetic)
  - ✓ Automatic garbage collection (can't explicitly free allocated memory)
  - ✓ Array bounds checking
  - ✓ Checking references for null

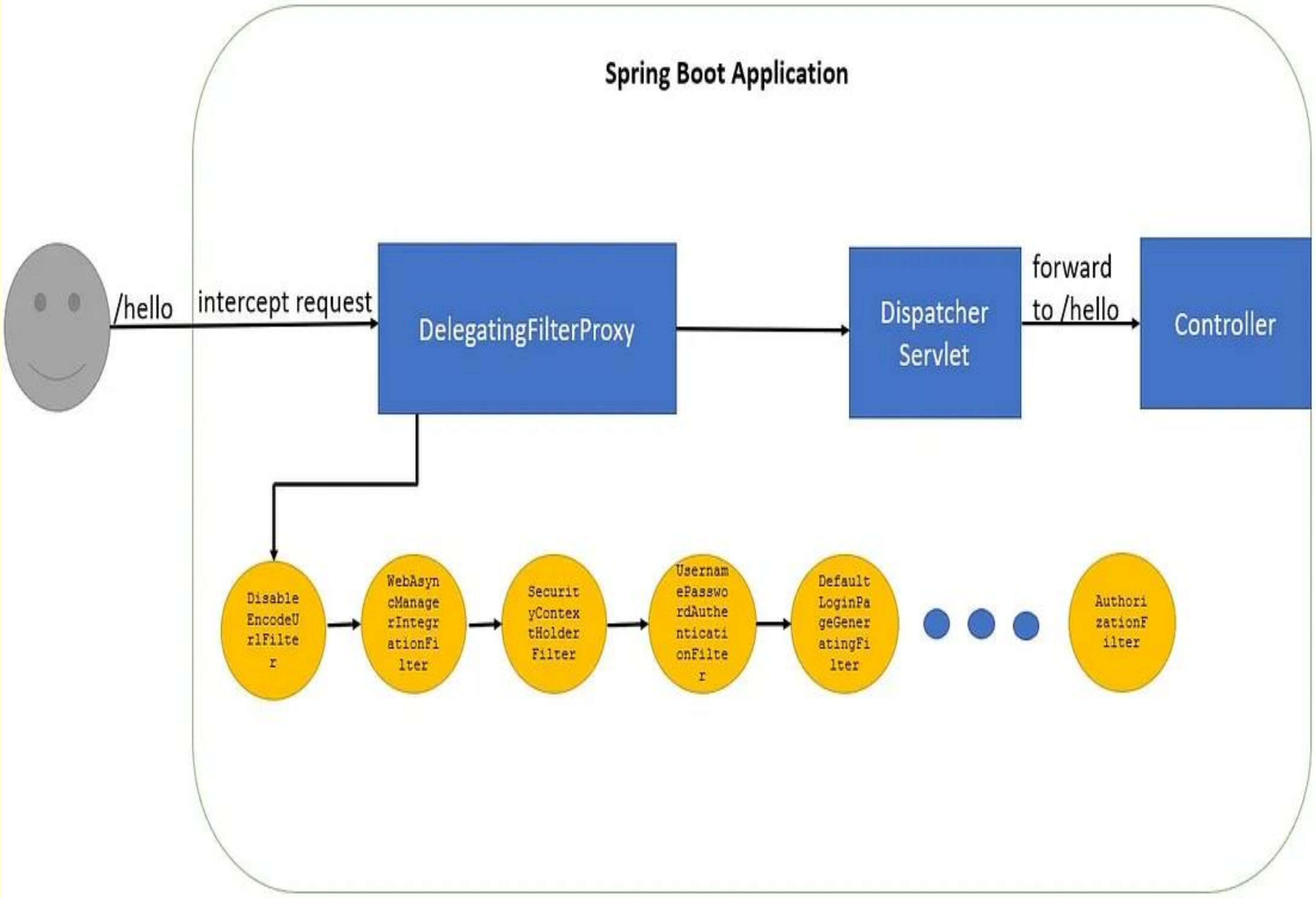
Java EE Security Model

- Security is something where its needed to take extra care, else the application will be vulnerable for attackers.
- The JavaEE security model is a **role–based, declarative model** based on **container–managed security**, where **resources** are protected by **roles** that are **assigned to users**.
- The model allows **decoupling** the **application** from its underlying **security infrastructure** since security can be specified separate from the application logic in an application D.D.
- The **container**, where application runs, **provides security** for the application according to a specifications in the D.D.
- This model also allows embedding security data (annotations) in the application code that can be referenced in deployment descriptors.

Diagrammatic Representation



- The core responsibility of a DispatcherServlet is to dispatch incoming HttpRequests to the correct handlers. Once we add the spring security dependency in the application, spring security gets enabled by default.


Spring Boot Application
/hello
intercept request
DelegatingFilterProxy
Dispatcher Servlet
forward to /hello
Controller
DisableEncodeUrlFilter
WebAsyncManagerIntegrationFilter
SecurityContextHolderFilter
UsernamePasswordAuthenticationFilter
DefaultLoginPageGeneratingFilter
AuthorizationFilter

contd..

- The **default username** for the form login is **user** and the password is printed by the spring boot application on startup console
- When the spring security dependency is added, spring security enables the security filter chain. These filters are responsible for Spring Security.
- So any incoming request will go through these filters and it is here that authentication and authorization takes place.

contd..

- Spring Security maintains a filter chain internally where each of the filters has a particular responsibility and filters are added or removed from the configuration depending on which services are required.
- Based on the type of requests there are different Authentication Filters like the BasicAuthenticationFilter,
- UsernamePasswordAuthenticationFilter. The ordering of the filters is important as there are dependencies between them.
  There will be three scenarios here -

contd..

- All the above Filters get called from the FilterChainProxy class. For this example we will mostly look at UsernamePasswordAuthenticationFilter, DefaultLoginPageGeneratingFilter and ExceptionTranslationFilter.
- **UsernamePasswordAuthenticationFilter —** This filter checks if the login url contains '/login' and is a POST request or not. As this is a GET request with url '/hello' so this filter does nothing.
- **DefaultLoginPageGeneratingFilter —** This filter checks if for the incoming request any authentication is done or not. Also it checks if the url is /login, /logout or /login?error. If it is niether of these it does nothing. In our case the url is /hello so it does nothing.
- **AuthorizationFilter —** This is the last filter in the filter chain. As no other filter has completed the authentication process, so this Filter throws an AccessDeniedException
- **ExceptionTranslationFilter —** This filter catches the AccessDeniedException. It then forces the application to redirect the browser to call the /login request.

Role

- Role enables to provide for an access or withdraw **particular features in applications** for authenticated user.
- Role is an abstract name for a group of users.
- Role can be compared to a key that can open a lock. Many people might have a copy of the key. The lock doesn't care who you are, only that you have the right key.
- Spring Security provides several ways to check user roles in code.

  a. The first way to check for user roles in Java is to use the *@PreAuthorize* annotation can be applied to a class or method.

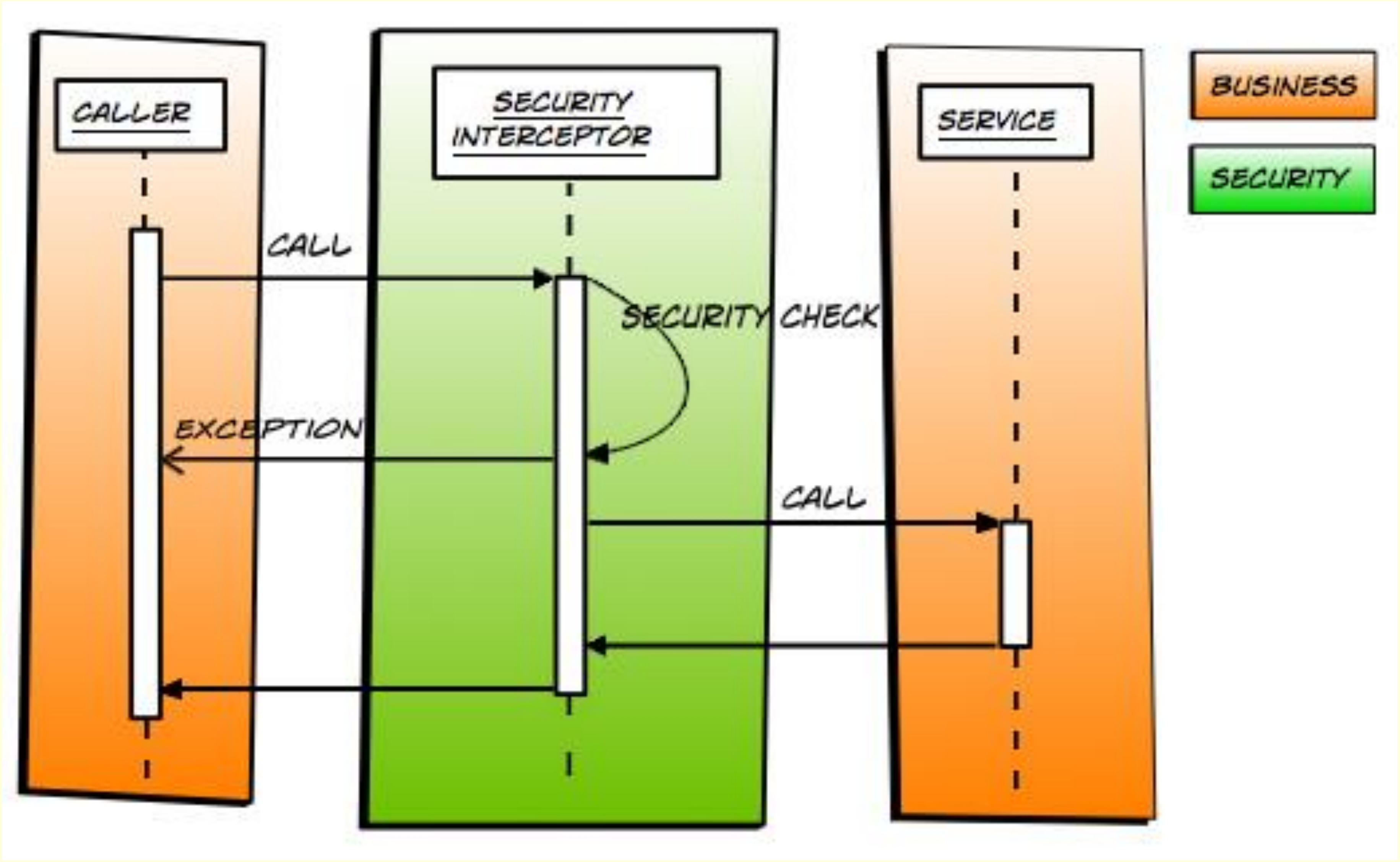
  b. The next way to check for user roles in Java code is with the *SecurityContext* class.

  c. The third way is by using the *UserDetailsService*. This bean can be injected anywhere in application and call it as needed:

Authentication and Authorization

- A server authenticates the end-user of a Web application in the following ways: **basic, form,** or **client-cert**.
- User–data i.e. **account–name** and **password** is processed by server to create a **credential** or to **deny access**. Server authentication is specified in the D.D.
- ***Security role* -** a set of users that can be specified in the application deployment descriptor or in the application code.
- Application controls **access to its resources** (such as a Web module URL or a bean method) by specifying the roles that are allowed to perform a given operation on a resource.
- During deployment, **application-scoped roles** are **mapped** to **security entities** in the running environment, such as **users, a group of users, or principals**. The mapping is specified in another **configuration file**.

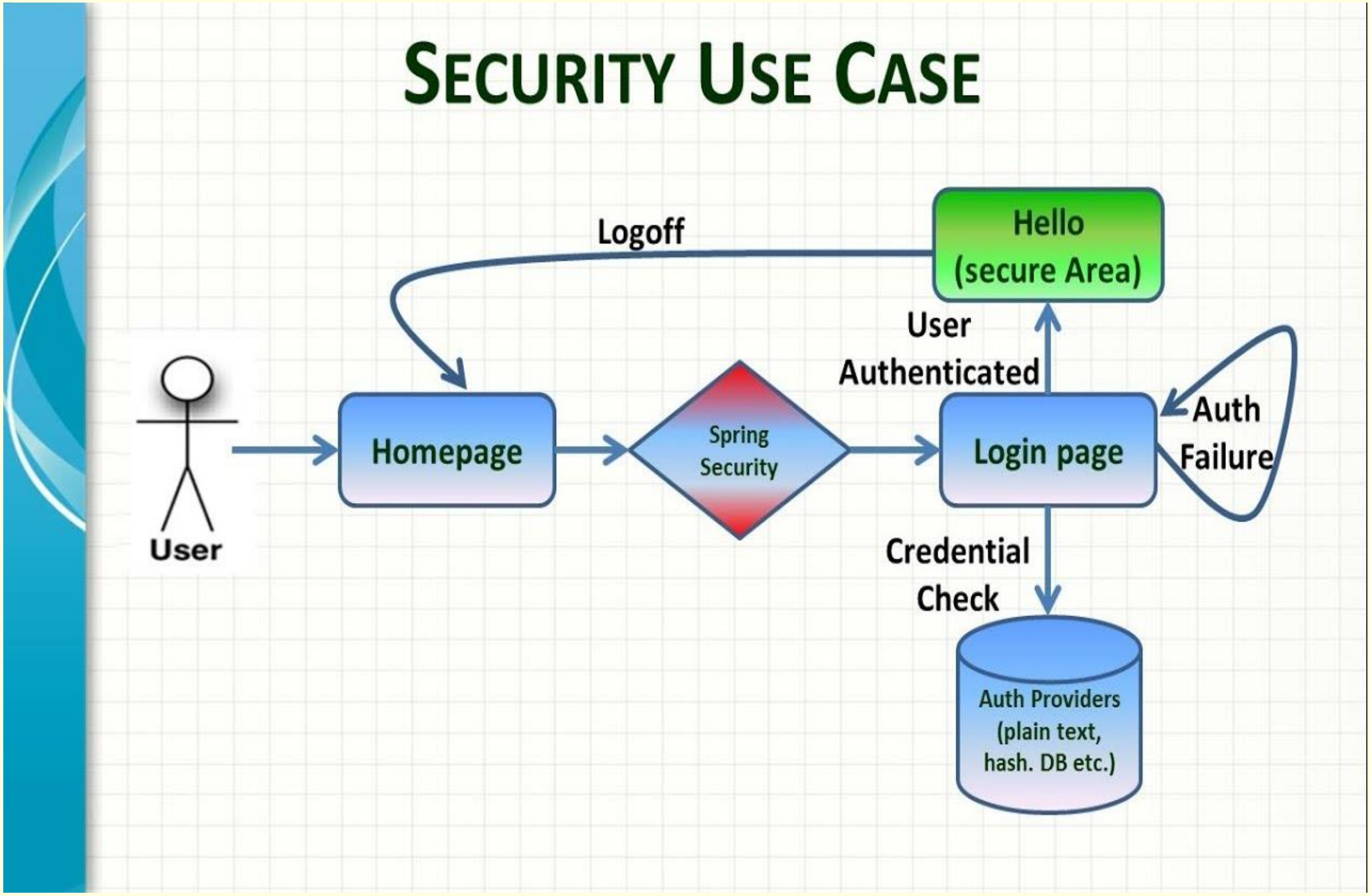
Diagrammatic Representation



Spring Security

- Spring Security is a framework that
  - ✓ is powerful, customizable for authentication & access-control.
  - ✓ provides ways to perform authentication and authorization.
  - ✓ is the de–facto standard for securing Spring–based applications.
  - ✓ integrates well with Spring MVC and comes bundled with popular security algorithm implementations.
  - ✓ is proven technology, better to use than reinvent the wheel.
- Spring security aims to resolve all the security issues that come during creating non-security Spring applications. It was **first released in 2008** as Spring Security 2.0.0.

contd..


SECURITY USE CASE
Logoff
Hello (secure Area)
User Authenticated
User
Homepage
Spring Security
Login page
Auth Failure
Credential Check
Auth Providers (plain text, hash. DB etc.)

contd..

- Features –

- ✓ **Easy–integration** with web application as Spring **auto–injects** security filters into the web application. Hence no need to modify web application configurations.
- ✓ **Comprehensive** and **extensible support** for both Authentication and Authorization, support for **groups** and **roles**.
- ✓ Protection against attacks like session fixation, clickjacking, cross site request forgery, etc.
- ✓ Provides support for authentication by different ways –

  in–memory, DAO, JDBC, LDAP and many more.
- ✓ Provides option to ignore specific URL patterns, good for serving static HTML, image files.

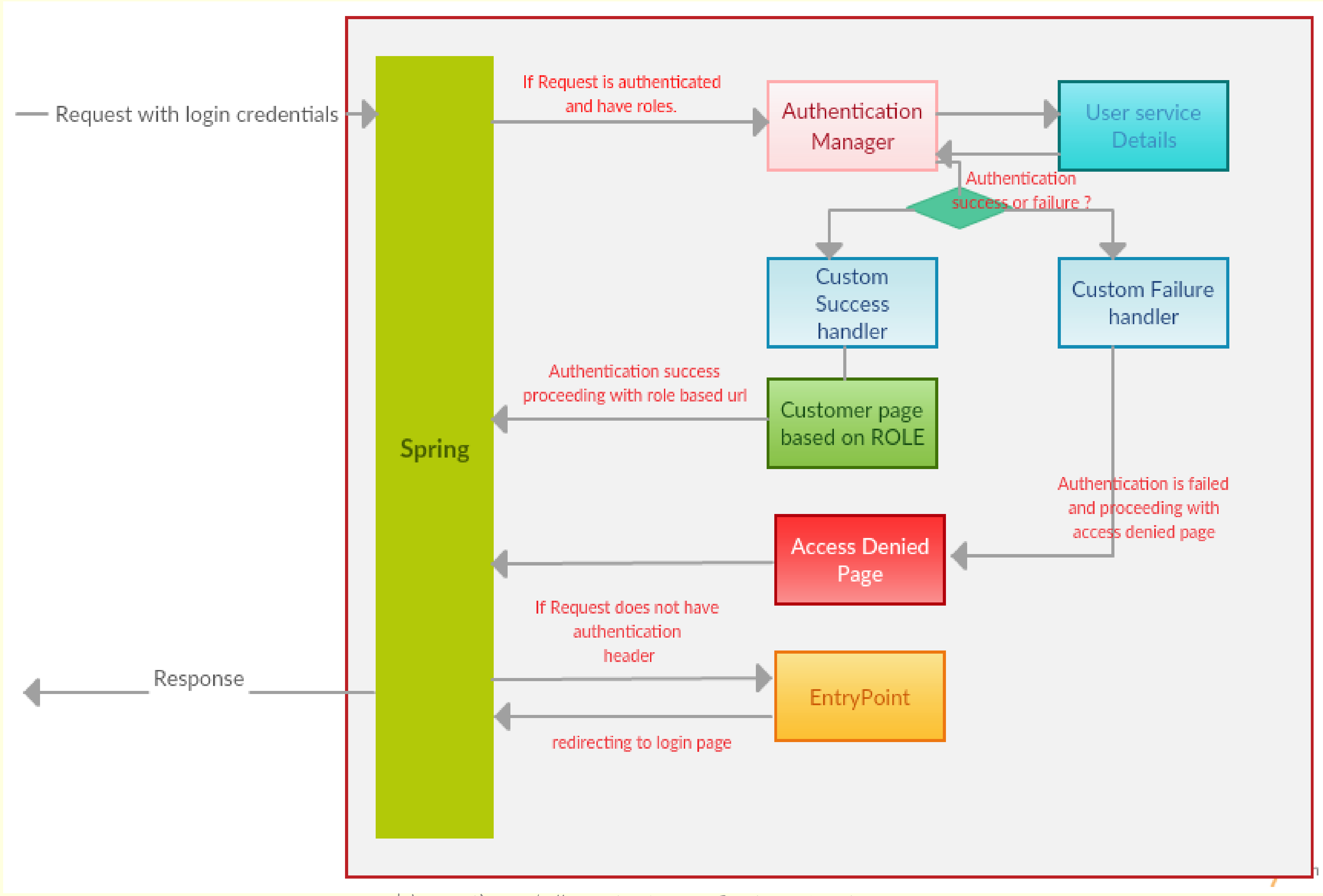
contd..

- Application security is **two more or less independent problems**
  - ✓ authentication means " who are you? " and
  - ✓ authorization " what are you allowed to do? " .
  - ✓ these are sometimes referred as "access control".
  - ✓ **Authorization** is the process to allow authority to perform actions in the application viz.. to authorize web request, methods and access to individual domain.
- Spring Security has an architecture that is designed to separate authentication from authorization and has strategies and extension points for both.

Introduction

- Spring Security is a bunch of **servlet filters** that help developer add to web application, authentication and authorization.
- Any Spring web application is only one servlet:
  Spring's DispatcherServlet that **redirects** incoming HTTP requests (e.g. from a browser) to @Controller or @RestController.
- There is **no security hardcoded** into that DispatcherServlet, its a raw HTTP Basic Auth header in @Controllers.
- Authentication and authorization should be done ***before*** the request hits @Controllers.

Diagrammatic Representation


Request with login credentials
Spring
If Request is authenticated and have roles.
Authentication Manager
User service Details
Authentication success or failure ?
Custom Success handler
Custom Failure handler
Authentication success proceeding with role based url
Customer page based on ROLE
Authentication is failed and proceeding with access denied page
Access Denied Page
If Request does not have authentication header
EntryPoint
redirecting to login page
Response

prepared by- Vijay Kulkarni, Java Spring Trainer

Spring Security Modules

- Spring Security is made of several independent modules such as core, auth, test, etc. Below listed are some module names:

- **Core:** It includes Spring Security's core classes and interfaces related to authentication and application access control.
- **Remoting:** It is used for handling the Spring Remoting application and contains the necessary classes.
- **Aspect:** It is used to include Aspect-Oriented Programming (AOP) support within Spring Security.
- **Config:** It is used to configure the Spring Security application by using XML and Java.
- **Crypto:** This module contains classes and interfaces for cryptography support.

contd..

- **Data:** It is used to integrate Spring Security with Spring Data.
- **Messaging:** It is helpful to implement messaging in the application.
- **OAuth2:** It includes classes and interface for OAuth 2.x within Spring Security:
- **OpenID:** It provides support to integrate OpenID web-authentication.
- **CAS:** CAS (Central Authentication Service) client integration.
- **TagLib:** It contains several tag libraries regarding Spring Security.
- **Test:** It adds testing support in the Spring Security.
- **Web:** It contains web security code, such as filters and Servlet API dependencies.

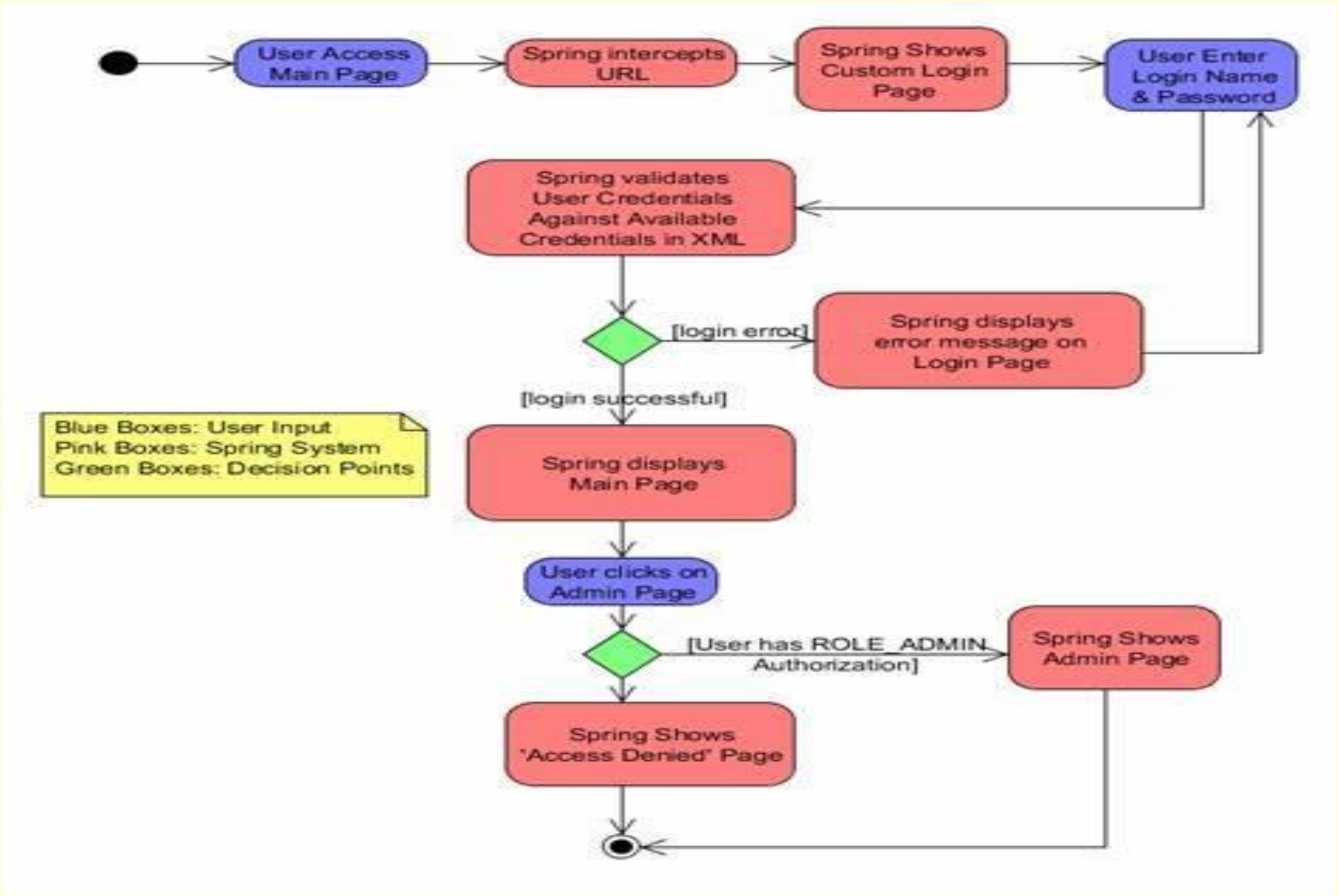
Spring Security features

- LDAP (Lightweight Directory Access Protocol) – It is an open application protocol for maintaining and accessing distributed directory information services over an Internet Protocol.
- Single sign-on – This feature allows a user to access multiple applications with the help of single account(user name and password).
- JAAS (Java Authentication and Authorization Service) LoginModule – It is a Pluggable Authentication Module implemented in Java.
- Basic Access Authentication – Spring Security supports Basic Access Authentication that is used to provide user name and password while making request over the network.
- Reactive Support – Version 5.0 onwards, it provides reactive programming and reactive web runtime support and can be integrated with Spring WebFlux.

contd..

- Digest Access Authentication – This feature asks browser to confirm user-identity before sending sensitive data on network.
- Remember-me – Spring Security remembers to the user and avoid login again from the same machine until the user logs out.
- Authorization-Spring Security provides this feature to authorize the user before accessing resources.
- Software Localization – This feature allows us to make application user interface in any language.
- HTTP Authorization – Spring provides this feature for HTTP authorization of web request URLs using Apache Ant paths or regular expressions
- OAuth 2.0 Login – This feature from ver 5.0 provides the facility to the user to login into the application by using their existing account at GitHub or Google.
- Modernized Password Encoding – A new Password encoder was introduced **DelegatingPasswordEncoder** which is more modern.

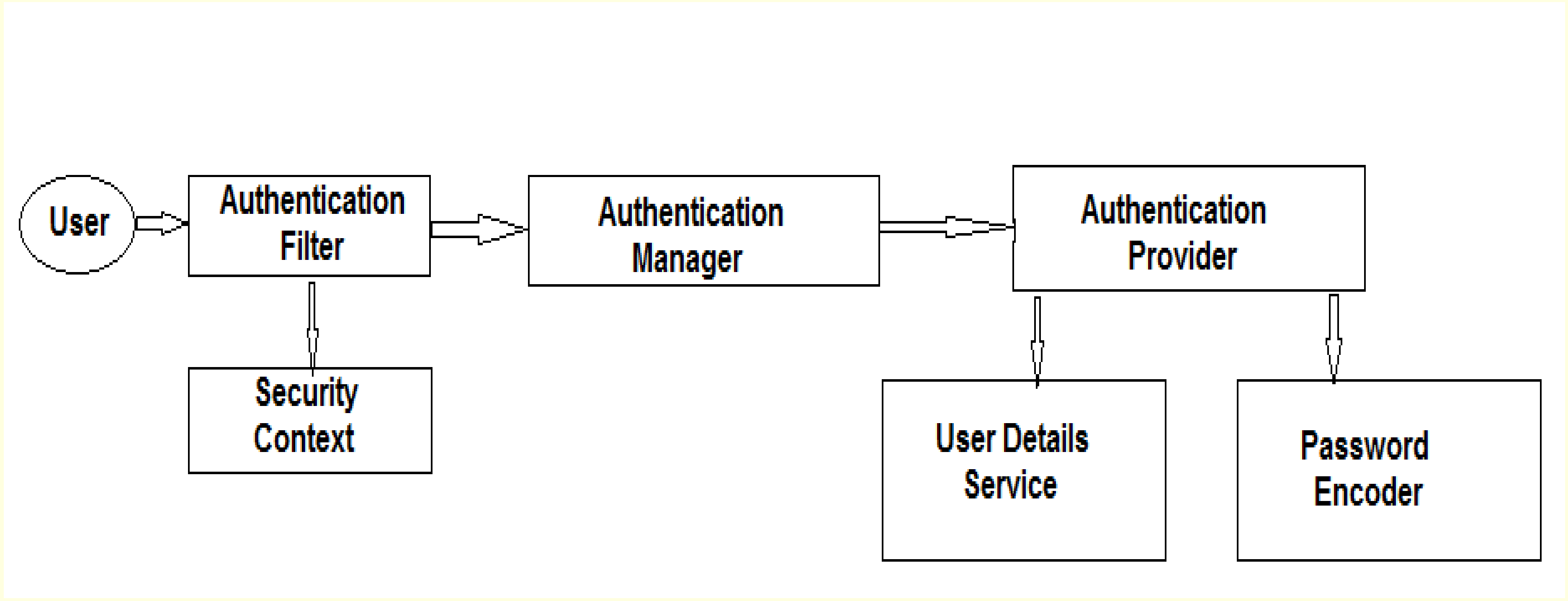
contd..


User Access Main Page
Spring intercepts URL
Spring Shows Custom Login Page
User Enter Login Name & Password
Spring validates User Credentials Against Available Credentials in XML
[login error]
Spring displays error message on Login Page
[login successful]
Blue Boxes: User Input
Pink Boxes: Spring System
Green Boxes: Decision Points
Spring displays Main Page
User clicks on Admin Page
[User has ROLE_ADMIN Authorization]
Spring Shows Admin Page
Spring Shows "Access Denied" Page

contd..

- The Security module is divided into separate jar files. The purpose was to divide jar files based on their functionalities. This helps to set required dependency into pom.xml file of maven project.
- The following are the jar files that are included into Spring Security module.
  - ✓ spring-security-core.jar
  - ✓ spring-security-remoting.jar
  - ✓ spring-security-web.jar
  - ✓ spring-security-config.jar
  - ✓ spring-security-ldap.jar
  - ✓ spring-security-oauth2-core.jar
  - ✓ spring-security-oauth2-client.jar
  - ✓ spring-security-oauth2-jose.jar
  - ✓ spring-security-acl.jar
  - ✓ spring-security-cas.jar
  - ✓ spring-security-openid.jar
  - ✓ spring-security-test.jar

Spring Security Components



- **Authentication filter:** The authentication filter receives user requests and forwards them to the authentication manager for authentication.
- **Authentication Manager:** The authentication manager uses the authentication provider to perform the authentication process.

contd..

- **Authentication Provider:** The authentication provider validates if the user exists in the system using the User details service and validates the password using the password encoder.
- **User details service:** Implements user management responsibility. It searches for the existing user, which is used in the authentication of the user by the provider.
- **Password Encoder:** Encodes the password before saving and matches the encoded password against a encode type.
- **Security Context:** The security context holds the user details after the successful authentication and authorization process.
- Spring security is a highly customizable framework. Any part of the framework can be customized. The framework configuration, such as authentication logic, custom representation of the user details, etc.

Spring Security 6 with Spring Boot 3

- Spring Security is a framework that provides authentication, authorization and protection against common attacks.
- Provides first class support for securing both web and reactive applications.
- Defacto standard for securing Spring applications.
- **spring-boot-starter-security** jar file that aggregates Spring security related dependencies together.
- Deprecated dependencies –
    1. WebSecurityConfigurerAdapter
    2. EnableGlobalMethodSecurity
    3. antMatchers
    4. authorizeRequests

Spring Boot Auto Configuration for Spring Security

- Spring boot auto configures below features
- spring-boot-starter-security starter aggregates Spring security related dependencies
- Enables Spring Security's default configuration which creates a servlet Filter as springSecurityFilterChain. Provides a default form.
- Creates a default user with username as **user** and a randomely generated password to the console.
- Spring Boot provides properties to customize default user's username and password
- Protects the password storage with Bcrypt algorithm
- Lets the user log out (default logout feature)
- CSRF attack prevention(enabled by default). So no additional code is necessary.
- If Spring Security is on classpath, Spring Boot automatically secures all HTTP endpoints with "basic" authentication.

Annotations

- @EnableWebMvc – @EnableWebMvc is used to enable Spring MVC is equivalent to <mvc:annotation-driven /> in XML. It enables support for @Controller-annotated classes that use @RequestMapping to map incoming requests to a certain method.
- @EnableWebSecurity – The @EnableWebSecurity is a marker annotation. It allows Spring to find (it's a @Configuration and, therefore, @Component) and automatically apply the class to the global WebSecurity. @EnableWebSecurity is used for spring security java configuration. Usage of this annotation is to add with @Configuration on top of the security java class that extends WebSecurityConfigurerAdapter.

Interfaces

- WebMvcConfigurer – Defines callback methods to customize the Java-based configuration for Spring MVC enabled via @EnableWebMvc . @EnableWebMvc – annotated configuration classes may implement this interface to be called back and given a chance to customize the default configuration.
- The *UserDetailsService* interface is used to retrieve user-related data. It has one method named *loadUserByUsername()* which can be overridden to customize the process of finding the user.

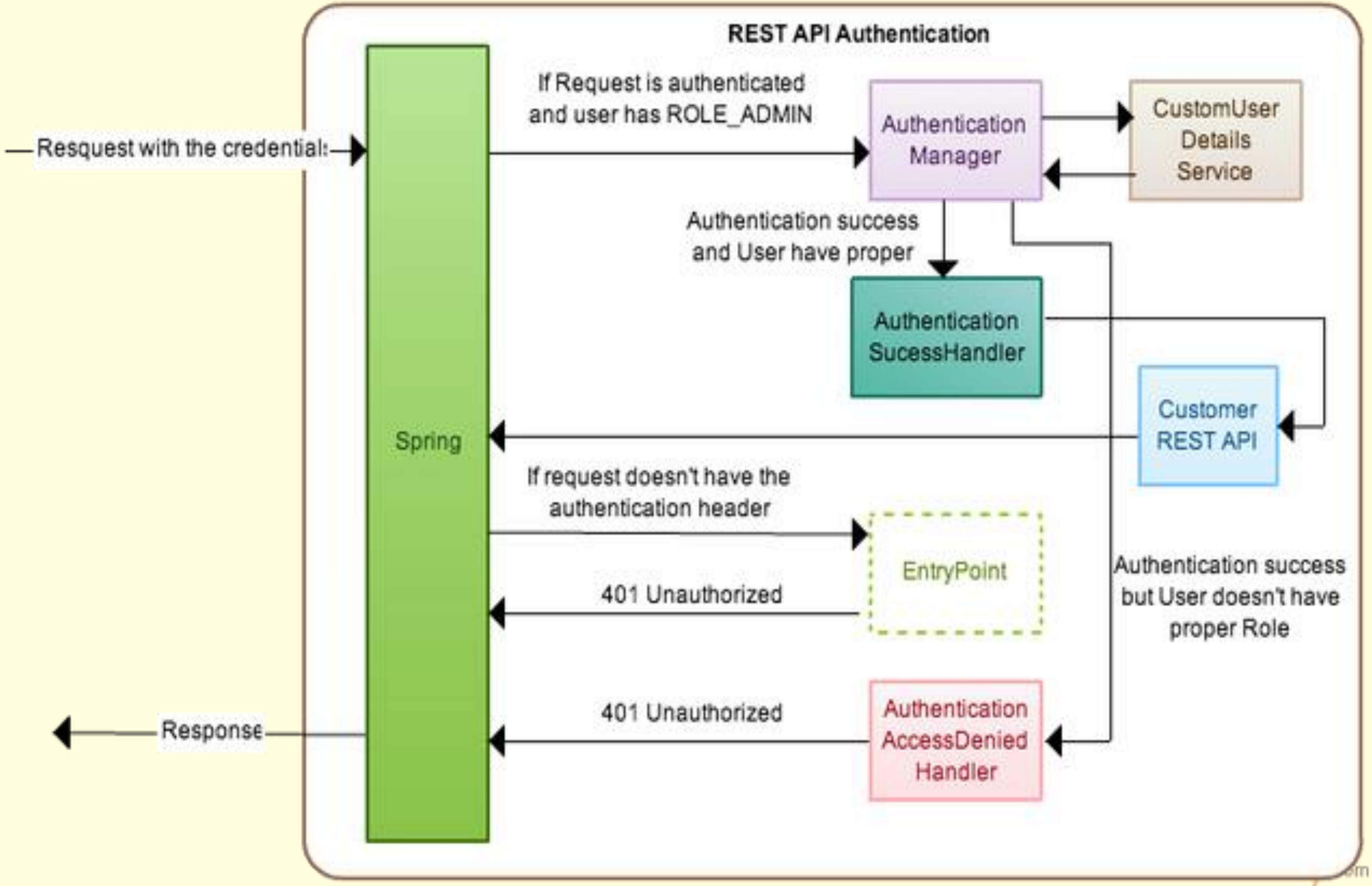
Classes

- Background – WebApplicationInitializer is used for booting Spring web applications. WebApplicationInitializer registers a Spring DispatcherServlet and creates a Spring web application context.
- AbstractAnnotationConfigDispatcherServletInitializer, which is an implementation of the WebApplicationInitializer, to create Spring web applications.
- Traditionally, Java web applications based on Servlets were using web.xml file to configure a Java web application. Since Servlet 3.0, created programatically via Servlet context listeners.
- InMemoryUserDetailsManager – Non-persistent implementation of UserDetailsManager which is backed by an in-memory map. Mainly intended for testing and demonstration purposes, where a full blown persistent system isn't required.

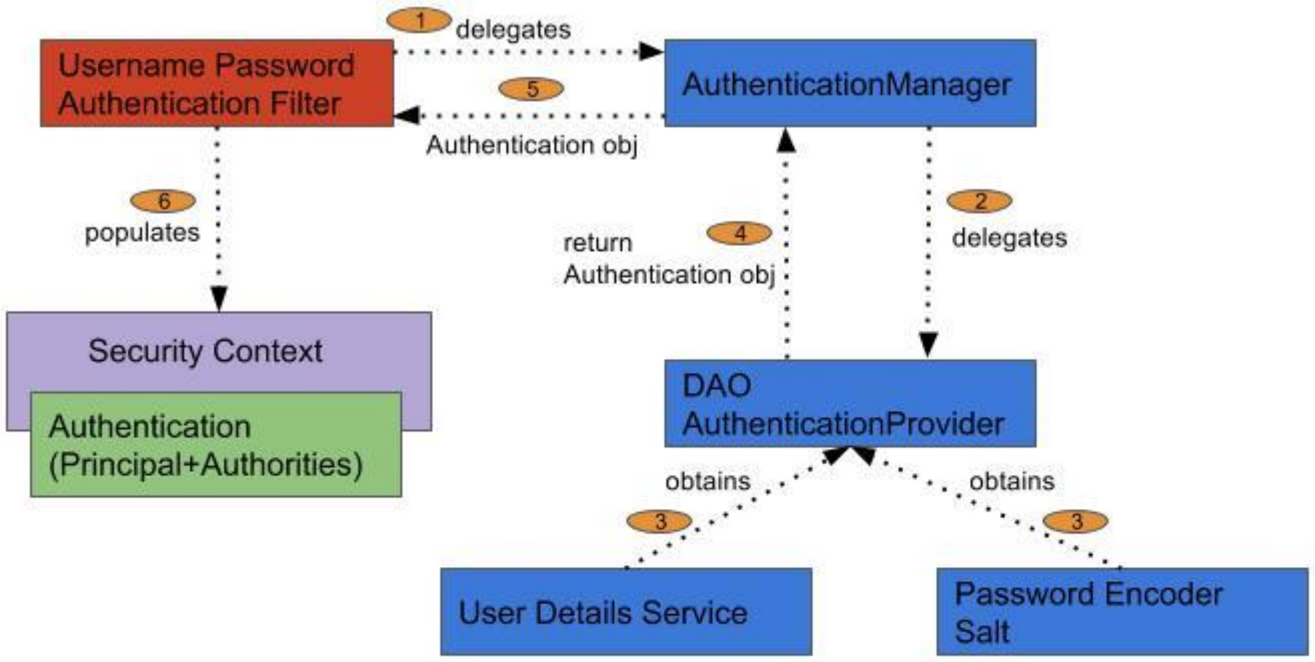
contd..

- AbstractAnnotationConfigDispatcherServletInitializer – a base class to initialize Spring application in Servlet container environment. It's an extension of WebApplicationInitializer. The subclasses are supposed to provide the classes annotated with @Configuration, Servlet config classes and DispatcherServlet mapping pattern.
- It is a newly(3.2+ versions of Spring) introduced Template method based, base class that makes Spring pure java based web application Configuration(@Configuration) without using a web.xml.
- HttpSecurity – It allows configuring web based security for specific http requests. By default it will be applied to all requests, but can be restricted using requestMatcher(Request Matcher) or other similar methods. Similar to Spring Security's XML <http> element.

contd..


REST API Authentication
Resquest with the credentials
Spring
If Request is authenticated and user has ROLE_ADMIN
Authentication Manager
CustomUser Details Service
Authentication success and User have proper
Authentication SucessHandler
Customer REST API
If request doesn't have the authentication header
EntryPoint
401 Unauthorized
Authentication success but User doesn't have proper Role
Authentication AccessDenied Handler
401 Unauthorized
Response

contd..

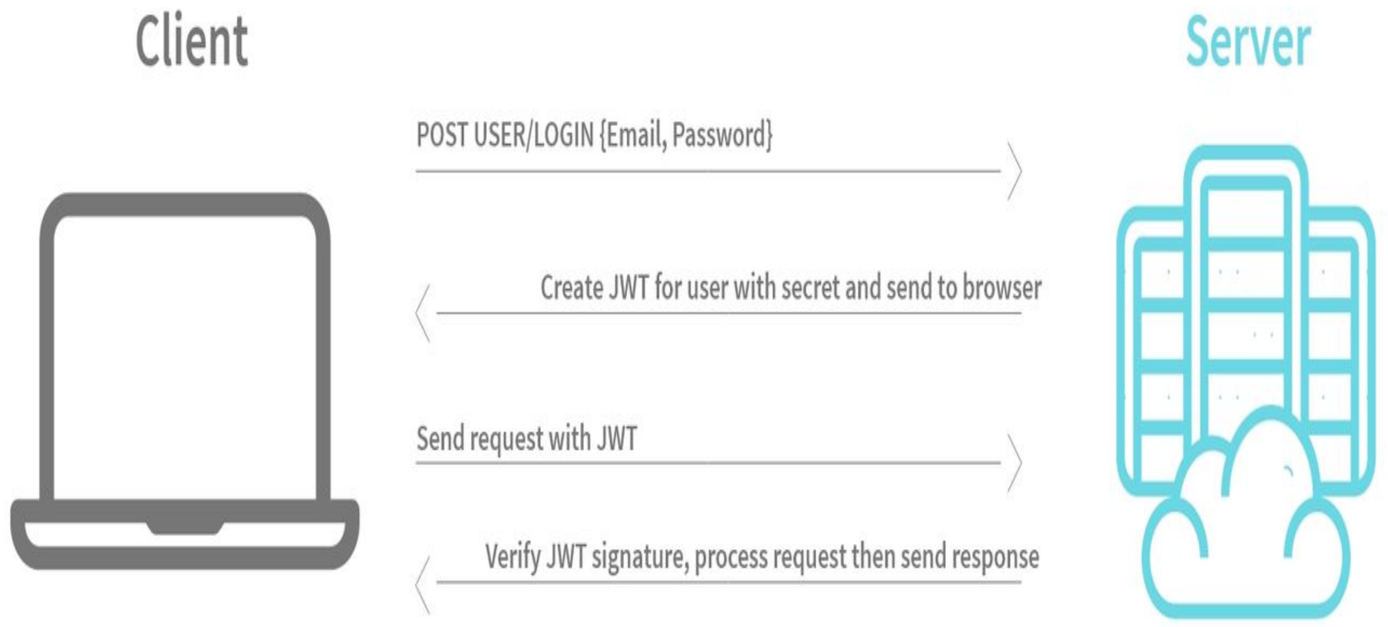

Username Password Authentication Filter
1
delegates
AuthenticationManager
5
Authentication obj
6
populates
return Authentication obj
4
2
delegates
Security Context
Authentication (Principal+Authorities)
DAO AuthenticationProvider
obtains
3
obtains
3
User Details Service
Password Encoder Salt

JWT

- **JSON Web Token (JWT)** is an **authentication method,** which allows for secure and scalable identity verification via stateless authentication.
- JWT is a stateless method of securely transmitting information between parties as a JavaScript Object Notation (JSON) object. Often used to **authenticate** and **authorize** users in web applications and APIs.
- In a stateless authentication system, each request is self-contained and includes all the necessary information to authenticate and authorize the user or entity. In the case of JWT authentication, it is token form.
- **JSON token** consists of **three** parts:

1. **Header** containing information about the type of token and algorithms used to generate the signature.
2. **Payload** containing **ID** and **authentication verifications** made by the user that can include a User ID, the user's name, an email address, and metainformation about the operation of the token.
3. A Signature, or cryptographic mechanism, is used to verify the token's integrity.

contd..

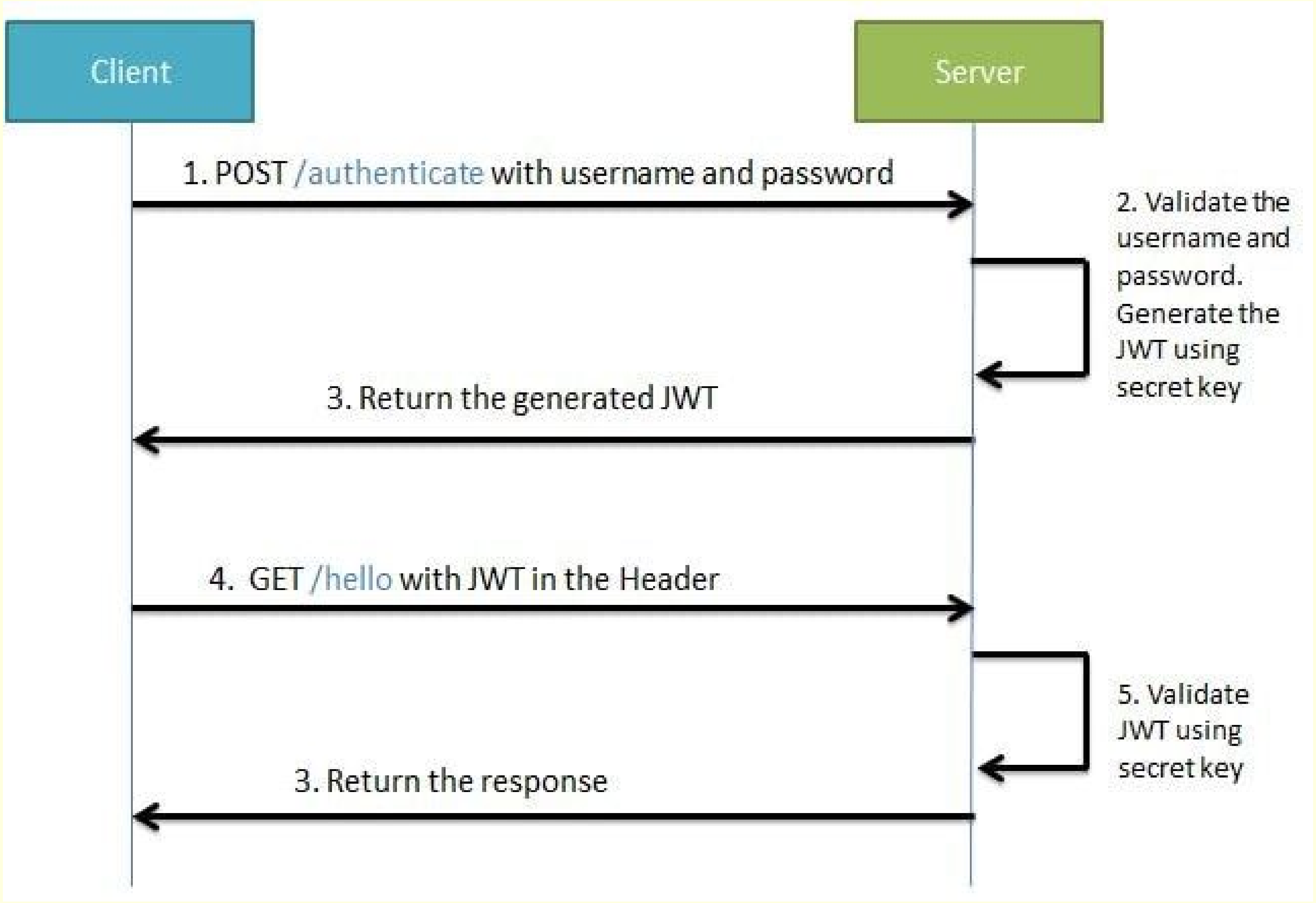

Client
Server
POST USER/LOGIN {Email, Password}
Create JWT for user with secret and send to browser
Send request with JWT
Verify JWT signature, process request then send response

contd..

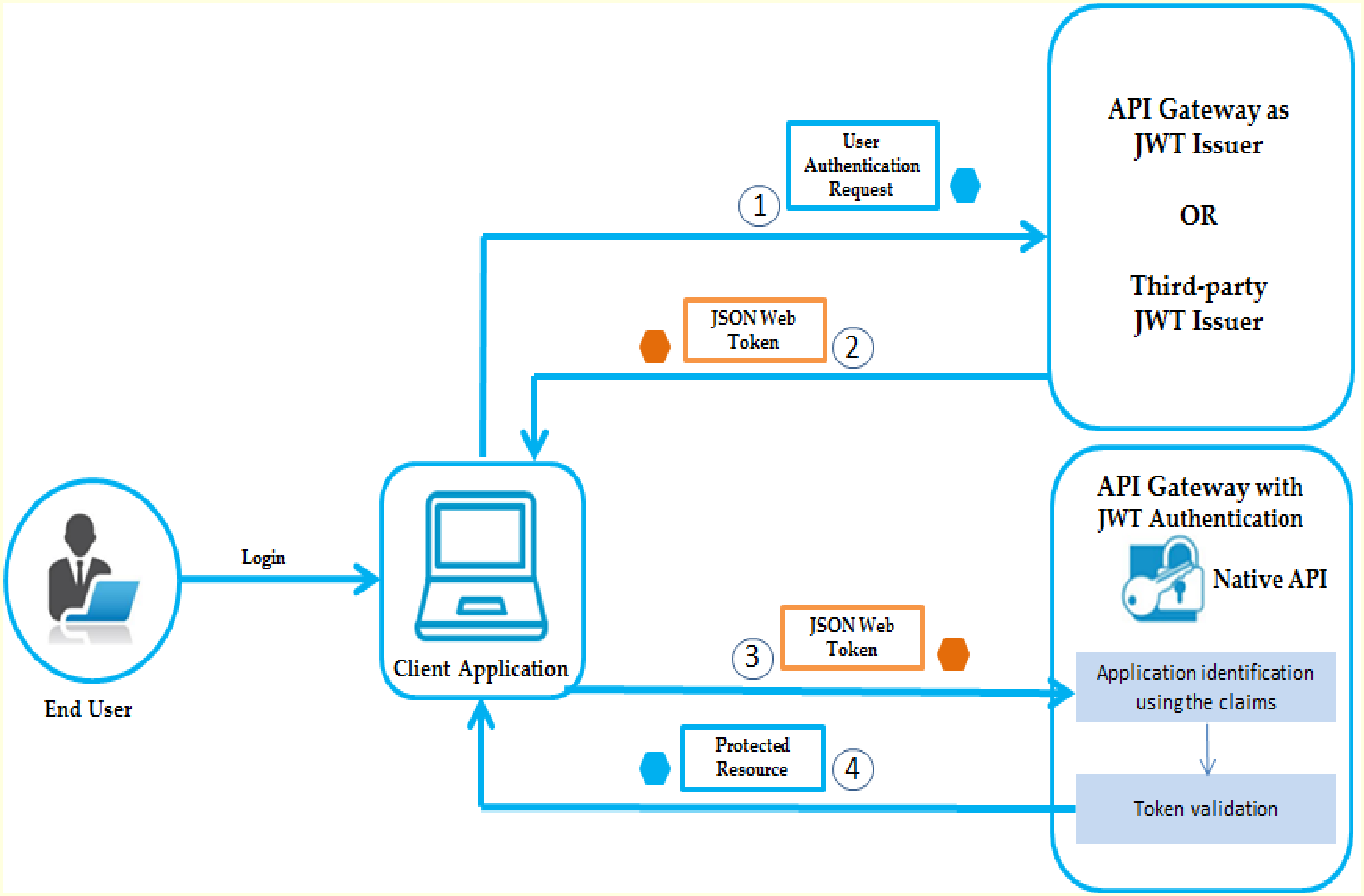
❖ How JWT authentication works –

1. **User Login:** The user provides their credentials (such as a username and password) to the web application or system for verification, which is transmitted to the authentication server.
2. **Token Generation**: Upon successful authentication, the server generates a JSON token containing critical information about the user and the authentication session. The server sends the token to the client for verification. When the JWT expires, the client must obtain a new JWT by logging in again.
3. **Token Storage**: The client stores the token, usually in a cookie or purpose-marked local storage, and includes it in subsequent requests.to the server.
4. **User Verification**: When the client sends a request to the application server, it verifies the signature in the token and checks the claims in the payload to ensure that the user can access the requested resource.
5. **Server Response**: If the JWT is valid and the user can access the requested resource.

contd..


Client
Server
1. POST /authenticate with username and password
2. Validate the username and password. Generate the JWT using secret key
3. Return the generated JWT
4. GET /hello with JWT in the Header
5. Validate JWT using secret key
3. Return the response

contd..


API Gateway as JWT Issuer
OR
Third-party JWT Issuer
User Authentication Request
1
JSON Web Token
2
End User
Login
Client Application
API Gateway with JWT Authentication
Native API
JSON Web Token
3
Application identification using the claims
Token validation
Protected Resource
4

contd..

1. Add Maven Dependencies
2. Configure MySQL Database
3. Create JPA Entities - User and Role (Many-to-Many Mapping)
4. Create Spring Data JPA Repositories
5. JWT Implementation Classes
6. Spring Security Implementation
7. Creating DTO classes - LoginDto and JwtAuthResponse
8. Creating a Service Layer
9. Controller Layer - Login REST API return JWT Token
10. Insert SQL Scripts
11. Testing using Postman
12. Implement Role-Based Authorization

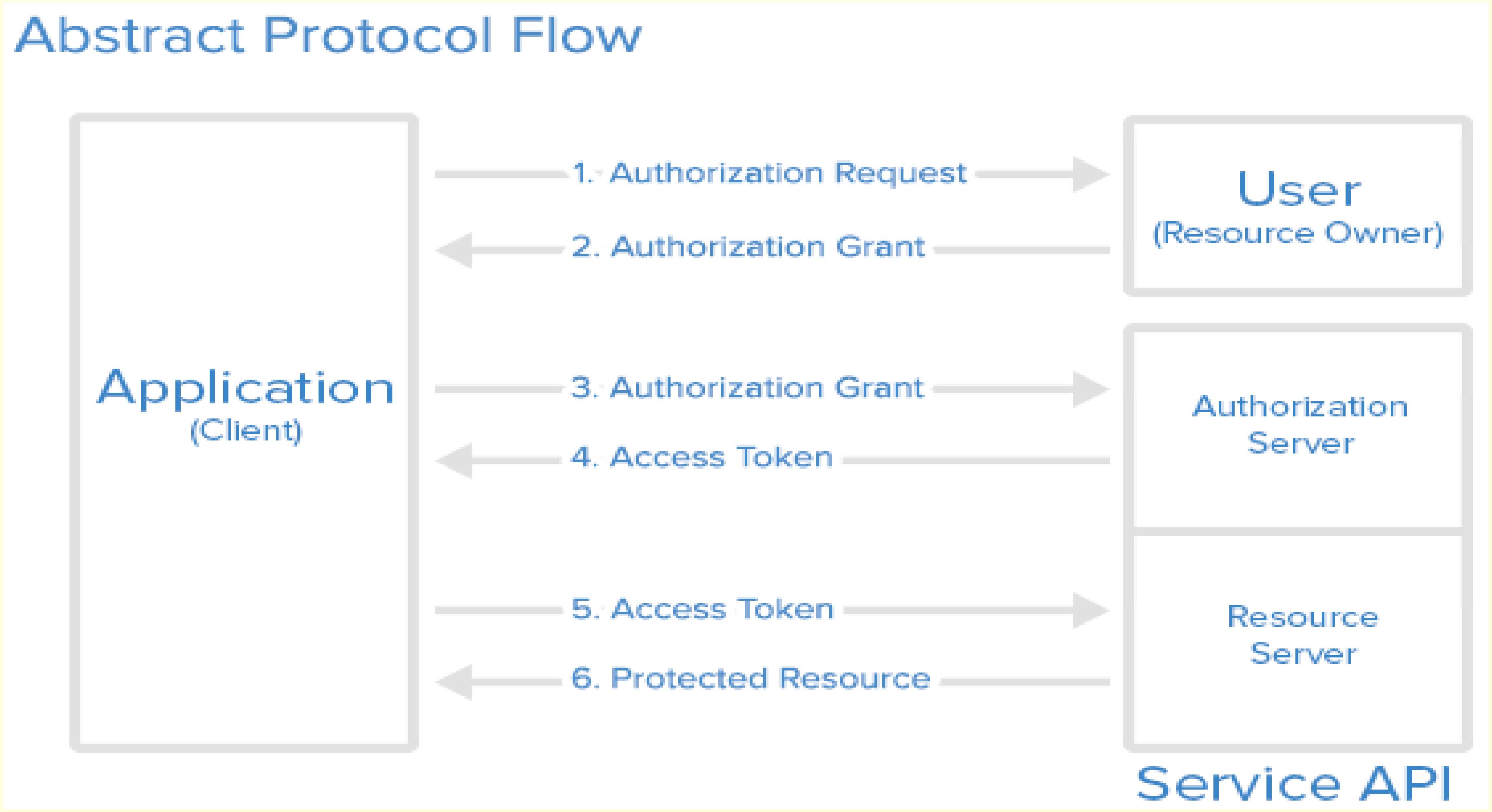
OAuth

- Scenario – A web service exists where user has registered. To access any resources on that service, the user must authenticate.
- Now, a third-party application wants to integrate with that service and present data to the user.
- The user could simply share his personal credentials with the third-party application. Results in user granting the third-party application unrestricted access to all their resources.
- There is also no way for the user (or the service) to control or revoke the third-party application's access.
- **OAuth** addresses this problem by separating the application's credentials from the user's credentials. Its designed to give a user control over what resources an application may or may not access on behalf of the user without having to share credentials.

contd..

- **OAuth** is an **Open Authorization protocol** that allows user to approve one application interacting with another on user's behalf without giving away user's password.
- OAuth uses **Access Tokens.** – An **Access Token** is a **piece of data** that represents the authorization to access resources on behalf of the end-user.
- OAuth doesn't define a specific format for Access Tokens. However, in some contexts, the JSON Web Token (JWT) format is often used.
- OAuth doesn't share password data but instead uses **authorization tokens** to prove an identity between **consumers** and **service providers.**

contd..

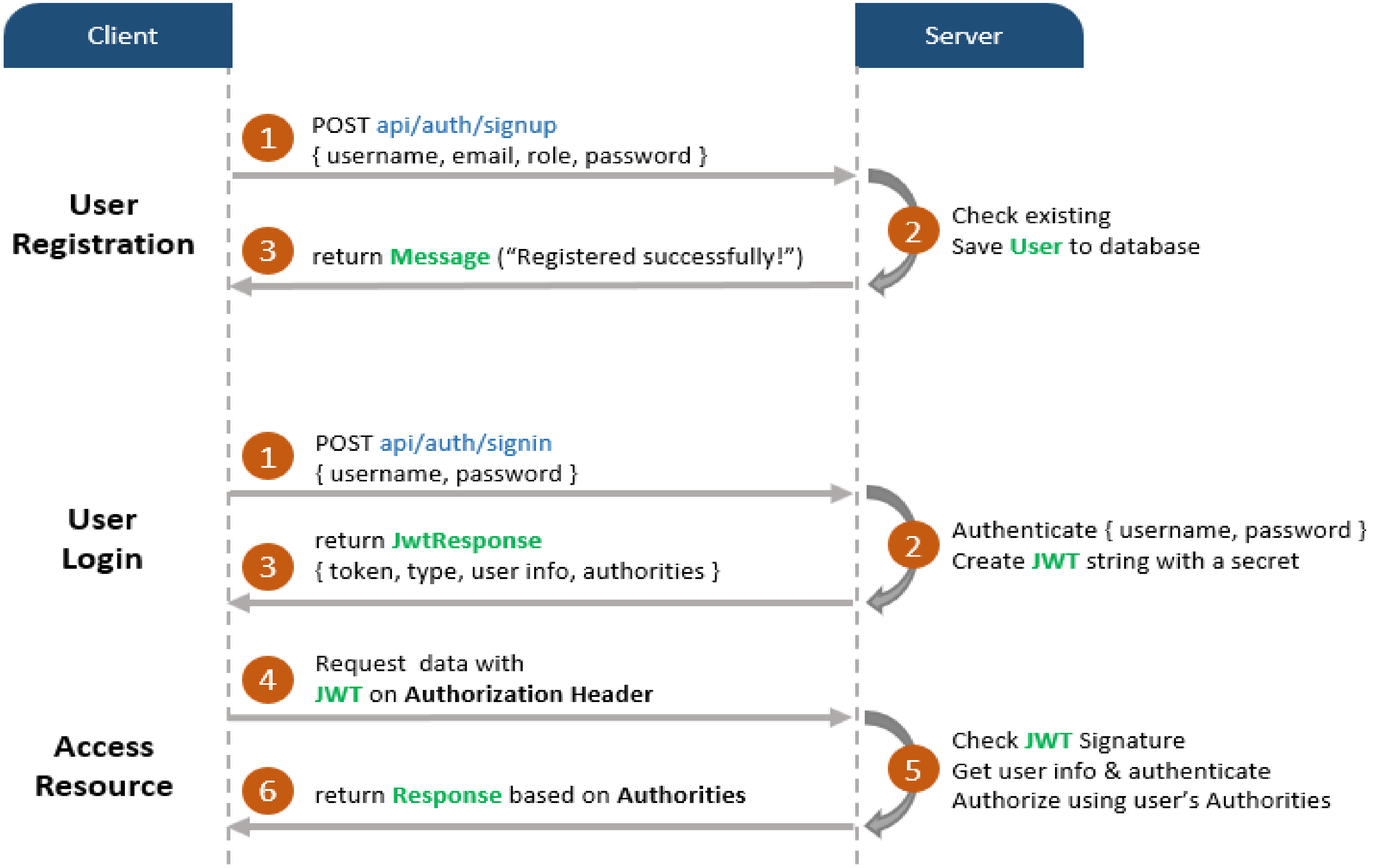

Abstract Protocol Flow
Application
(Client)
1. Authorization Request
2. Authorization Grant
3. Authorization Grant
4. Access Token
5. Access Token
6. Protected Resource
User
(Resource Owner)
Authorization Server
Resource Server
Service API

contd..

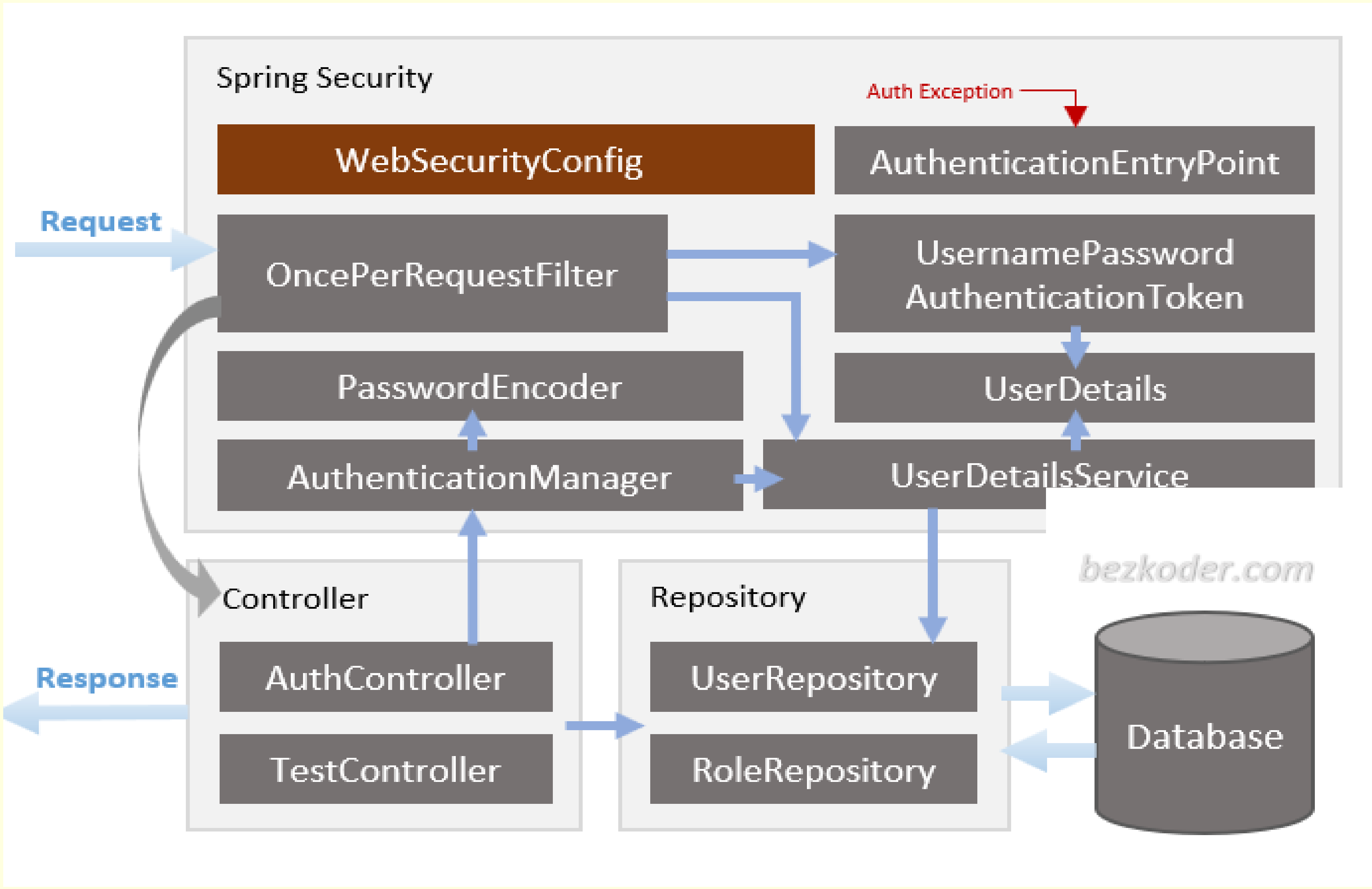
- The formal roles within OAuth are:

1. **The resource owner**: The entity granting access to some resources, often the end-user.
2. **The resource server**: The entity hosting the protected resources, such as an API. The resource server requires and validates access tokens as part of processing requests.
3. **The (OAuth) client**: The application that requests access to protected resources on behalf of the resource owner. It uses the access token to prove authorization.
4. **The authorization server**: The entity that authenticates the resource owner, obtains authorization, and issues access tokens to the client.

contd..


Client
Server
User Registration
1 POST api/auth/signup
{ username, email, role, password }
2 Check existing
Save User to database
3 return Message ("Registered successfully!")
User Login
1 POST api/auth/signin
{ username, password }
2 Authenticate { username, password }
Create JWT string with a secret
3 return JwtResponse
{ token, type, user info, authorities }
Access Resource
4 Request data with
JWT on Authorization Header
5 Check JWT Signature
Get user info & authenticate
Authorize using user's Authorities
6 return Response based on Authorities

contd..



Spring Security
Auth Exception
WebSecurityConfig
AuthenticationEntryPoint
Request
OncePerRequestFilter
UsernamePassword AuthenticationToken
PasswordEncoder
UserDetails
AuthenticationManager
UserDetailsService
bezkoder.com
Controller
Repository
Response
AuthController
UserRepository
Database
TestController
RoleRepository

contd..

- **WebSecurityConfig** – is the crux of our security implementation. It configures cors, csrf, session management, rules for protected resources. Can also extended and customized.
- **UserDetailsService** interface has a method to load User by *username* and returns a UserDetails object that Spring Security can use for authentication and validation.
- **UserDetails** contains necessary information (such as: username, password, authorities) to build an Authentication object.
- **UsernamePasswordAuthenticationToken** gets {username, password} from login request. AuthenticationManager will use it to authenticate a login account.
- **AuthenticationManager** has a DaoAuthenticationProvider (with help of UserDetailsService & PasswordEncoder) to validate UsernamePasswordAuthenticationToken object. If successful, AuthenticationManager returns a fully populated Authentication object (including granted authorities).

contd..

- OncePerRequestFilter makes a single execution for each request to the API. It provides a doFilterInternal() method that will implement parsing & validating JWT, loading User details (using UserDetailsService), checking Authorizaion (using UsernamePasswordAuthenticationToken).
- AuthenticationEntryPoint will catch authentication error.
- **Repository** contains UserRepository & RoleRepository to work with Database, will be imported into **Controller**.
- **Controller** receives and handles request after it was filtered by OncePerRequestFilter.
- AuthController handles signup/login requests
- TestController has accessing protected resource methods with role based validations.

Classes and Interfaces used in Project

- public interface **UserDetails** extends Serializable
- Provides core user information.
- Implementations are not used directly by Spring Security for security purposes. They simply store user information which is later encapsulated into Authentication objects. This allows non-security related user information (such as email addresses, telephone numbers etc) to be stored in a convenient location.
- Concrete implementations must take particular care to ensure the non-null contract detailed for each method is enforced.

contd..

- Collection<? extends GrantedAuthority> getAuthorities()
- Returns the authorities granted to the user.
- String getPassword() Returns the password used to authenticate the user.
- String getUsername() Returns the username used to authenticate the user.
- Boolean isAccountNonExpired() Indicates whether the user's account has expired.
- Boolean isAccountNonLocked() Indicates whether the user is locked or unlocked.
- Boolean isCredentialsNonExpired() Indicates whether the user's credentials (password) has expired.
- Boolean isEnabled() Indicates whether the user is enabled or disabled.

contd..

public final class **SimpleGrantedAuthority** extends Object implements GrantedAuthority

Basic concrete implementation of a GrantedAuthority.

Stores a String representation of an authority granted to the Authentication object.

contd..

public interface **GrantedAuthority** extends Serializable
Represents an authority granted to an Authentication object.
A GrantedAuthority must either represent itself as a String or be specifically supported by an AccessDecisionManager.

- ***Method Summary***

String
getAuthority()
If the GrantedAuthority can be represented as a String and that String is sufficient in precision to be relied upon for an access control decision by an AccessDecisionManager (or delegate), this method should return such a String.

Annotations

- The @PreAuthorize annotation **checks the given expression before entering the method**, whereas the
- @PostAuthorize annotation verifies it after the execution of the method and could alter the result.

Project - 2

- The major building blocks of Spring Security are:
- **SecurityContextHolder**, to provide access to the **SecurityContext**.
- **SecurityContext**, to hold the **Authentication** and possibly request-specific security information.
- **Authentication**, to represent the **principal** in a Spring Security-specific manner.
- **GrantedAuthority**, to reflect the application-wide permissions granted to a principal.
- **UserDetails**, to provide the necessary information to build an Authentication object from your application's DAOs or other source of security data.
- **UserDetailsService**, to create a UserDetails when passed in a String-based username (or certificate ID or the like).

Spring Security Example Application

- Spring MVC application that secures the page with a login form backed by a fixed list of users.
- The requirements of such an app could be:
- The app will have users, each with role **Admin** or **User**
- They log in by their **emails** and **passwords**
- Non-admin users can view their info, but cannot peek at other users
- Admin users can list and view all the users, and create new ones as well
- Customized form for login
- "**Remember me**" authentication for laziest
- Possibility to logout
- Home page will be available for everyone, authenticated or not

contd..

1. **Authentication** is an interface which has several implementations for different authentication models. For a simple user name and password authentication, spring security would use **UsernamePasswordAuthenticationToken**. When user enters username and password, system creates a new instance of **UsernamePasswordAuthenticationToken**.
2. The token is passed to an instance of **AuthenticationManager** for validation. Internally what **AuthenticationManager** will do is to iterate the list of configured **AuthenticationProvider** to validate the request. There should be at least one provider to be configured for the valid authentication.
3. The **AuthenticationManager** returns a fully populated Authentication instance on successful authentication.
4. The final step is to establish a security context by invoking **SecurityContextHolder.getContext().setAuthentication(),** passing in the returned authentication object.

Thank You

JWT

- Token-based authentication enables users to obtain a token that allows them to access a service and/or fetch a specific resource, without using their username and password to authenticate every request.
- A stateless system sends a request to the server and relays the response (or the state) back without storing any information. On the other hand, stateful systems expect a response, track information, and resend the request if no response is received.
- JSON Web Token (JWT) authentication is a stateless method of securely transmitting information between parties as a JavaScript Object Notation (JSON) object. It is often used to authenticate and authorize users in web applications and APIs.
- In a stateless authentication system, each request is self-contained and includes all the necessary information to authenticate and authorize the user or entity. In the case of JWT authentication, this comes in the form of a token. A JSON token consists of three parts:
- A Header containing information about the type of token and algorithms used to generate the signature.
- A Payload containing the “claims” (ID and authentication verifications) made by the user that can include a User ID, the user’s name, an email address, and metainformation about the operation of the token.
- A Signature, or cryptographic mechanism, is used to verify the token’s integrity.

contd..

- How JWT authentication works:

1. User Login: The user provides their credentials (such as a username and password) to the web application or system for verification, which is transmitted to the authentication server.
2. Token Generation: Upon successful authentication, the server generates a JSON token containing critical information about the user and the authentication session. The server sends the token to the client for verification.
3. Token Storage: The client stores the token, usually in a cookie or purpose-marked local storage, and includes it in subsequent requests to the server.
4. User Verification: When the client sends a request to the application server, it verifies the signature in the token and checks the claims in the payload to ensure that the user can access the requested resource.
5. Server Response: If the JWT is valid and the user can access the requested resource.

Thank You