Spring Security Masterclass

Faisal Memon (EmbarkX)

Usage Policy for Course Materials

Instructor: Faisal Memon
Company: EmbarkX.com

1. Personal Use Only
The materials provided in this course, including but not limited to PDF presentations, are intended for your personal use only. They are to be used solely for the purpose of learning and completing this course.

2. No Unauthorized Sharing or Distribution
You are not permitted to share, distribute, or publicly post any course materials on any websites, social media platforms, or other public forums without prior written consent from the instructor.

3. Intellectual Property
All course materials are protected by copyright laws and are the intellectual property of Faisal Memon and EmbarkX. Unauthorized use, reproduction, or distribution of these materials is strictly prohibited.

4. Reporting Violations
If you become aware of any unauthorized sharing or distribution of course materials, please report it immediately to [embarkxofficial@gmail.com].

5. Legal Action
We reserve the right to take legal action against individuals or entities found to be violating this usage policy.

Thank you for respecting these guidelines and helping us maintain the integrity of our course materials.

Contact Information
embarkxofficial@gmail.com
www.embarkx.com

Spring Security: Importance & Benefits

Faisal Memon (EmbarkX)

Spring Security

- Authentication and Authorization
- Protection Against Common Threats
- Password Storage
- Integration with Spring Ecosystem

Imagine doing all by yourself

Why Use Spring Security

→ *Comprehensive and Customizable*

→ *Community and Support*

→ *Declarative Security*

→ *Integration Capabilities*

Why Use Spring Security

→ *Regular Updates*

→ *Ease of Use with Spring Boot*

Thank you

PRINCIPAL AND AUTHENTICATION OBJECT

Faisal Memon (EmbarkX)

Principal

Principal *represents the currently logged-in user. Your user details (like your username or email) become your* ***Principal***

Authentication Object

Authentication Object *is a more comprehensive representation of the user's authentication information*

Principal: john_doe

Authorities: ROLE_ADMIN

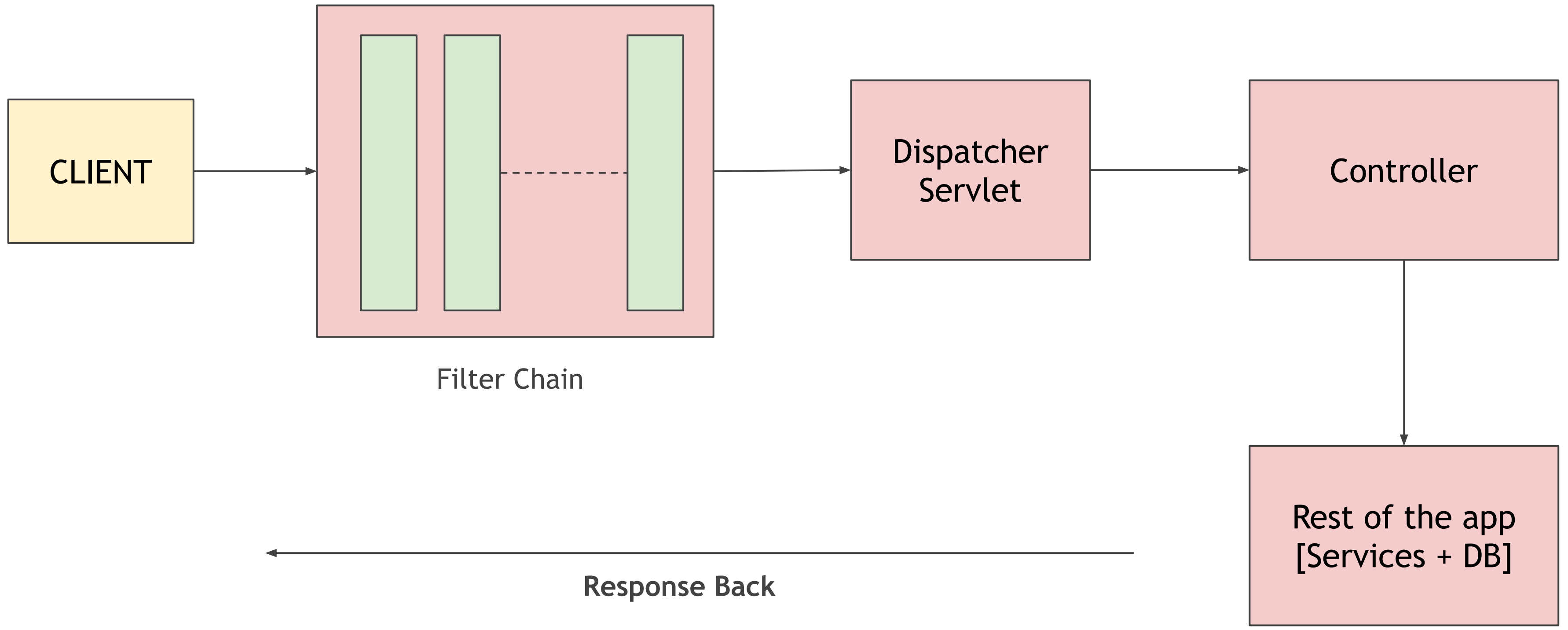
Authentication Object

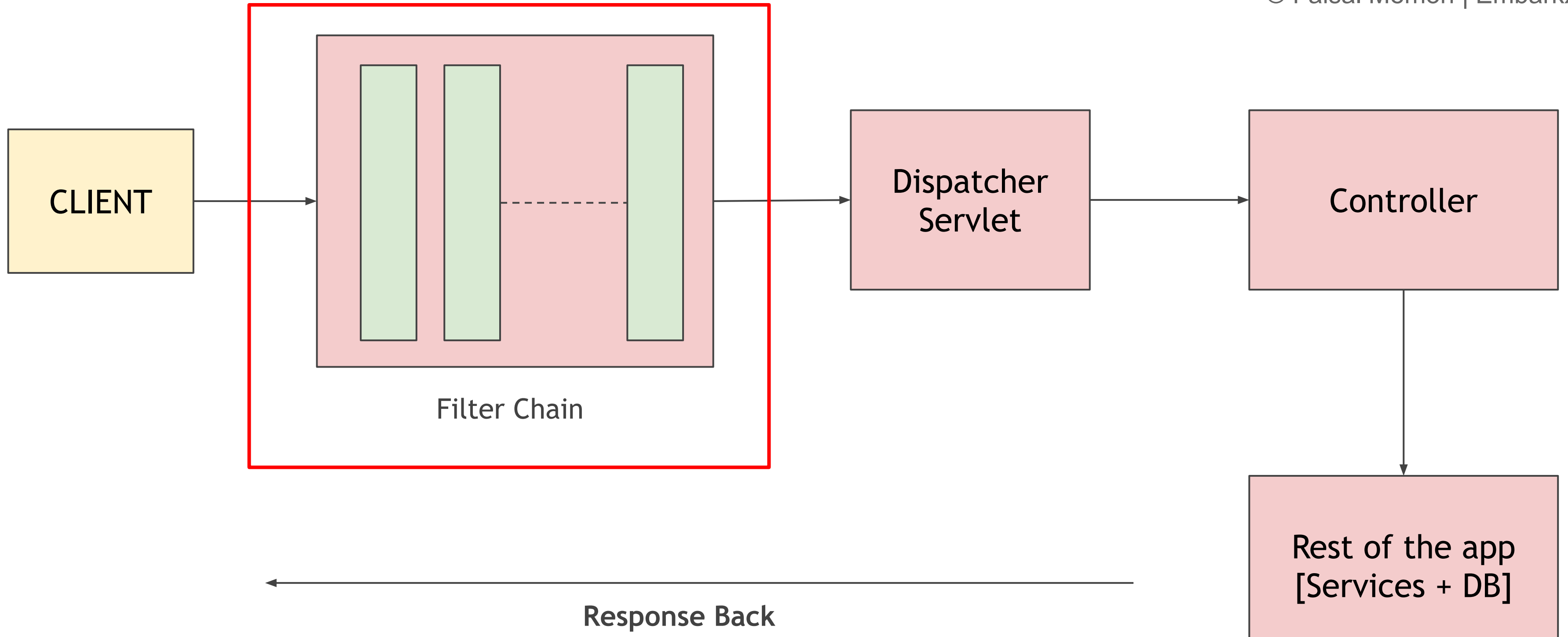
→ Who you are and what you can do

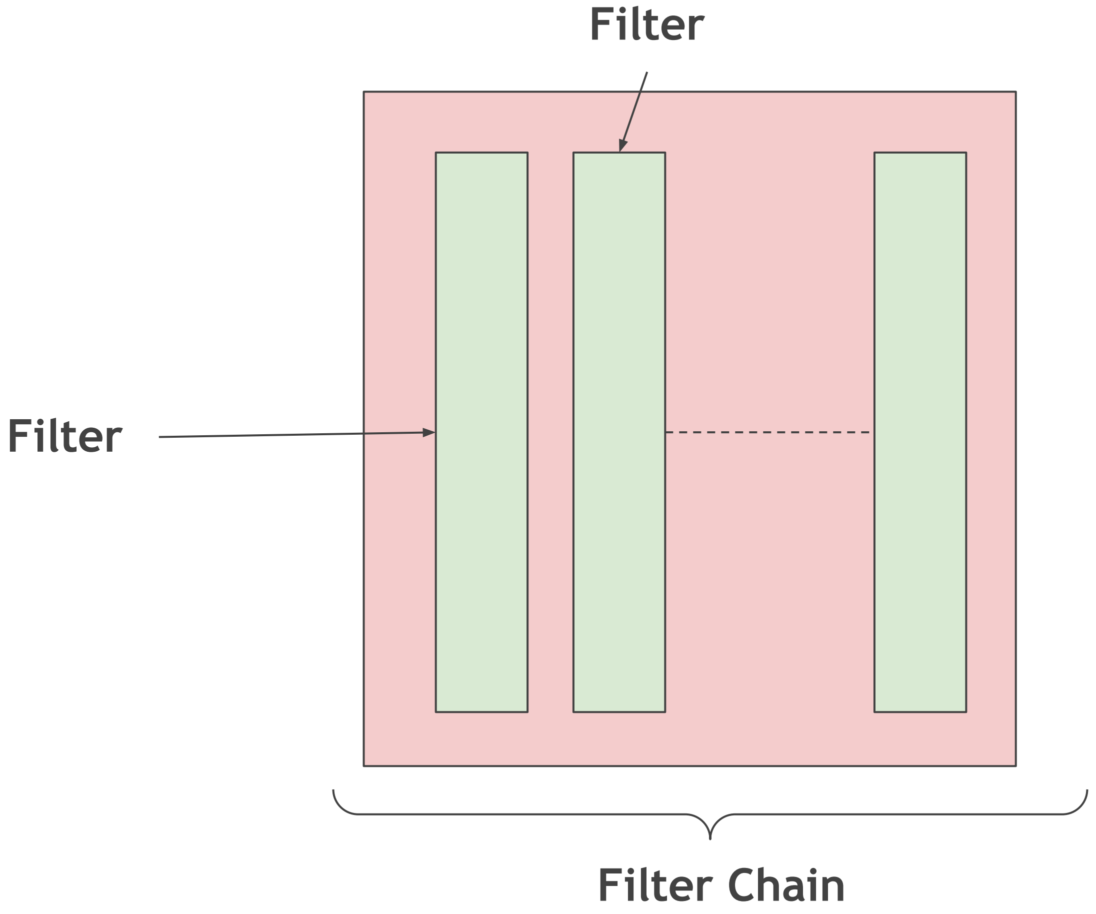
→Details about the user

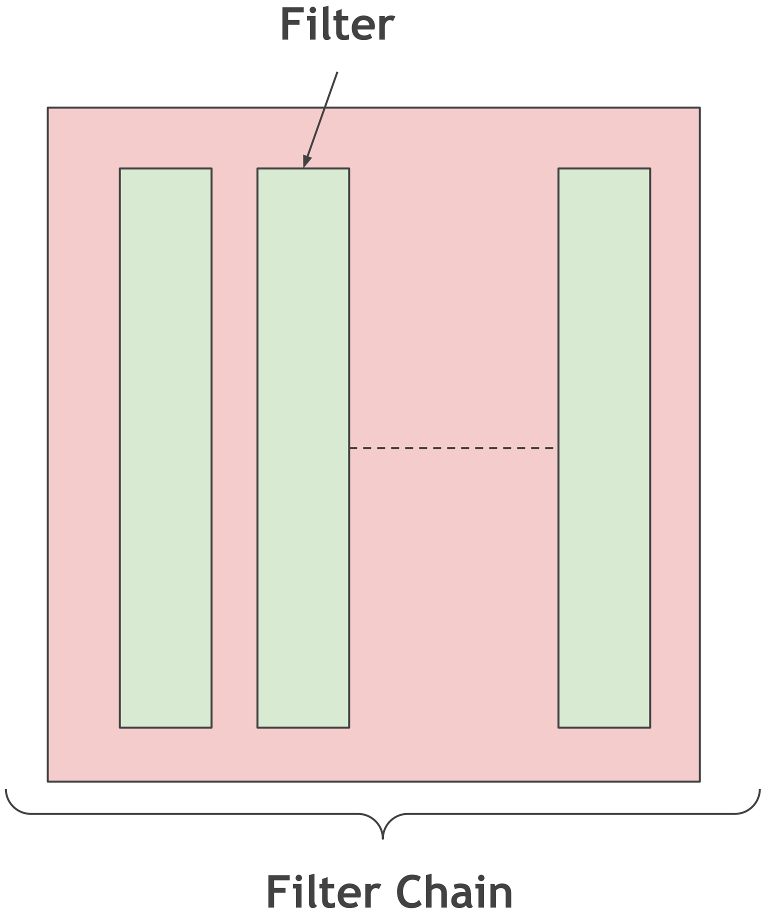
Understanding Filters and Filter Chain

Faisal Memon (EmbarkX)


CLIENT
Filter Chain
Dispatcher Servlet
Controller
Rest of the app [Services + DB]
Response Back


CLIENT
Filter Chain
Dispatcher Servlet
Controller
Rest of the app [Services + DB]
Response Back


Filter
Filter
Filter Chain



→ **Filters** are components that can intercept and modify incoming requests and outgoing responses in a web application.

→ A **Filter Chain** is a sequence of filters that an HTTP request and response pass through before reaching the targeted resource and after the resource has generated a response.

How Filters and Filter Chains work?

→ The first filter in the chain receives the request and performs its processing.

*→ After processing, the filter calls **chain.doFilter(request, response)** to pass the request to the next filter in the chain.*

→ This process continues until the request reaches the final resource

→ The response generated by the resource then travels back through the chain, allowing each filter to perform any necessary post-processing.

Summary

→ *Filters are components that can intercept and modify requests and responses.*

→ *Filter Chains are sequences of filters through which requests and responses pass.*

→ *In Spring Security, filters are used for authentication, authorization, and other security tasks, arranged in a chain managed by the framework.*

Why Filters?

→ *Cross-Cutting Concerns*

→ *Pre-Processing and Post-Processing*

→ *Request and Response Manipulation*

→ *Separation of Concerns*

Thank you

Filters that you should be aware of

Faisal Memon (EmbarkX)

Key Filters

SecurityContextPersistenceFilter

Manages the SecurityContext for each request.

Class: org.springframework.security.web.context.SecurityContextPersistenceFilter

WebAsyncManagerIntegrationFilter

Integrates the SecurityContext with Spring's WebAsyncManager for asynchronous web requests.

Class: org.springframework.security.web.context.request.async.WebAsyncManagerIntegrationFilter

Key Filters

HeaderWriterFilter

Adds security-related HTTP headers to the response, such as X-Content-Type-Options, X-Frame-Options, and X-XSS-Protection.

Class: org.springframework.security.web.header.HeaderWriterFilter

CorsFilter

Handles Cross-Origin Resource Sharing (CORS) by allowing or denying requests from different origins based on configured policies.

Class: org.springframework.web.filter.CorsFilter

Key Filters

CsrfFilter

Enforces Cross-Site Request Forgery (CSRF) protection by generating and validating CSRF tokens for each request.

Class: org.springframework.security.web.csrf.CsrfFilter

LogoutFilter

Manages the logout process by invalidating the session, clearing cookies, and redirecting the user to a configured logout success URL.

Class: org.springframework.security.web.authentication.logout.LogoutFilter

Key Filters

UsernamePasswordAuthenticationFilter

Processes authentication requests for username and password credentials. It handles the form-based login process.

Class: org.springframework.security.web.authentication.UsernamePasswordAuthenticationFilter

DefaultLoginPageGeneratingFilter

Generates a default login page if no custom login page is provided.

Class: org.springframework.security.web.authentication.ui.DefaultLoginPageGeneratingFilter

Key Filters

DefaultLogoutPageGeneratingFilter

Generates a default logout page if no custom logout page is provided.

Class: org.springframework.security.web.authentication.ui.DefaultLogoutPageGeneratingFilter

BasicAuthenticationFilter

Handles HTTP Basic authentication by extracting credentials from the Authorization header and passing them to the authentication manager.

Class: org.springframework.security.web.authentication.www.BasicAuthenticationFilter

Key Filters

RequestCacheAwareFilter

Ensures that the original requested URL is cached during authentication, so that the user can be redirected to it after successful authentication.

Class: org.springframework.security.web.savedrequest.RequestCacheAwareFilter

SecurityContextHolderAwareRequestFilter

Wraps the request to provide security-related methods (e.g., isUserInRole and getRemoteUser) that interact with the SecurityContext.

Class: org.springframework.security.web.servletapi.SecurityContextHolderAwareRequestFilter

Key Filters

AnonymousAuthenticationFilter

Provides anonymous authentication for users who are not authenticated. This is useful to apply security constraints even to unauthenticated users.

Class: org.springframework.security.web.authentication.AnonymousAuthenticationFilter

ExceptionTranslationFilter

Translates authentication and access-related exceptions into appropriate HTTP responses, such as redirecting to the login page or sending a 403 Forbidden status.

Class: org.springframework.security.web.access.ExceptionTranslationFilter

Key Filters

FilterSecurityInterceptor

Enforces security policies (authorization checks) on secured HTTP requests. It makes final access control decisions based on the configured security metadata and the current Authentication.

Class: org.springframework.security.web.access.intercept.FilterSecurityInterceptor

Why Learning these is important

→ *Demonstrates In-Depth Knowledge*

→ *Problem-Solving Skills*

→ *Security Best Practices*

Helps with Right Configuration, Customization and Troubleshooting

Thank you

Basic Authentication

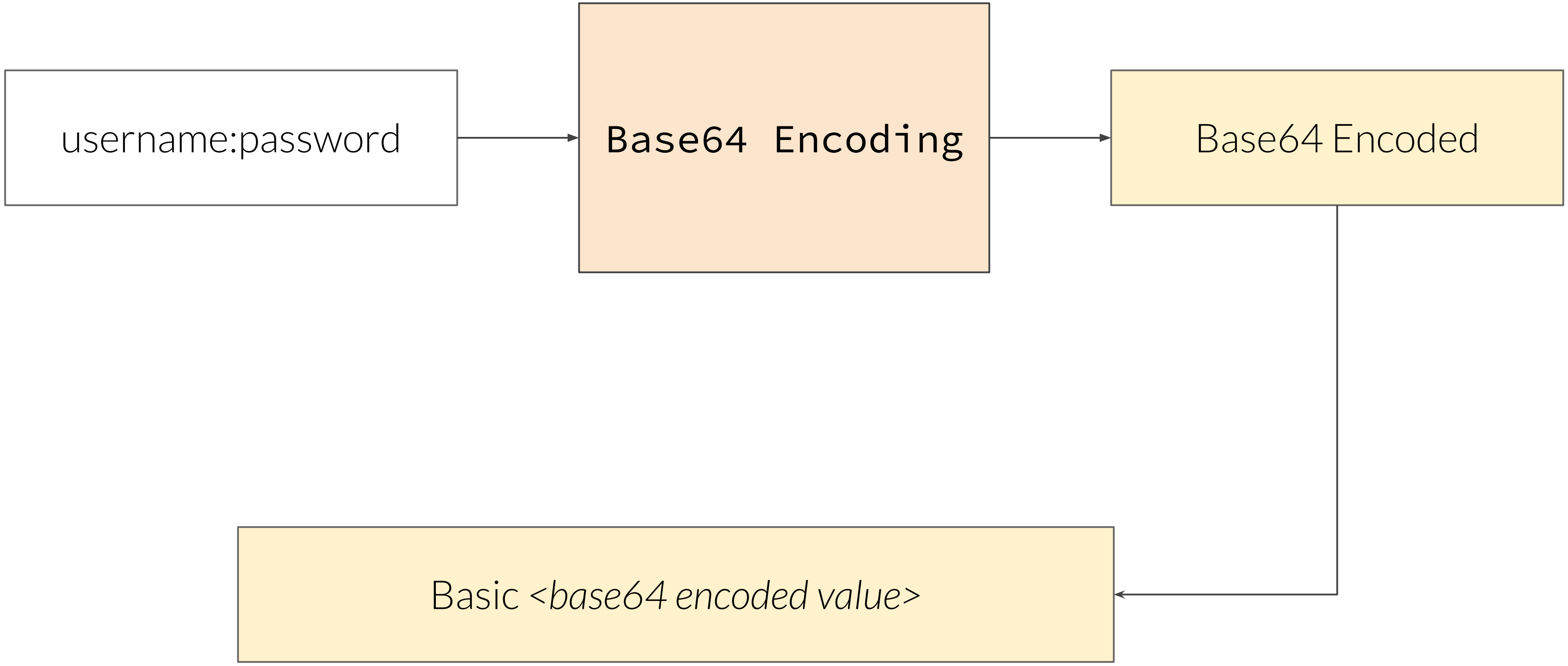
Faisal Memon (EmbarkX)

Basic Authentication

→ *Basic Authentication is one of the simplest forms of authentication supported by Spring Security*

→ *It involves sending the username and password with each HTTP request in the Authorization header.*

→ *The credentials are encoded using Base64 and sent over the network. Spring Security then decodes and validates these credentials.*



Authorization Header

Structuring Thoughts

Faisal Memon (EmbarkX)

HTTP Method	Endpoint	Description	Request Body	Request Parameter	Response
POST	/api/notes	Create a new note	`String content`	`@AuthenticationPrincipal UserDetails userDetails`	`Note` (created note)
GET	/api/notes	Retrieve all notes for the logged-in user	None	`@AuthenticationPrincipal UserDetails userDetails`	`List<Note>` (user's notes)
PUT	/api/notes/{noteId}	Update an existing note	`String content`	`@AuthenticationPrincipal UserDetails userDetails`	`Note` (updated note)
DELETE	/api/notes/{noteId}	Delete a note	None	`@AuthenticationPrincipal UserDetails userDetails`	`void`

Authentication Providers

Faisal Memon (EmbarkX)

Authentication Providers in Spring Security are components handle the actual verification of credentials provided by a user during the login process

Key Responsibilities

→ *Authenticate the User*

→ *Create Authentication Token*

Importance

→ *Flexibility*

→ *Separation of Concerns*

→ *Extensibility*

→ *Security*

Thank you

Authentication Providers

Faisal Memon (EmbarkX)

Authentication Providers

→ *DaoAuthenticationProvider*

→ *InMemoryAuthenticationProvider*

→ *LdapAuthenticationProvider*

→ *ActiveDirectoryLdapAuthenticationProvider*

Authentication Providers

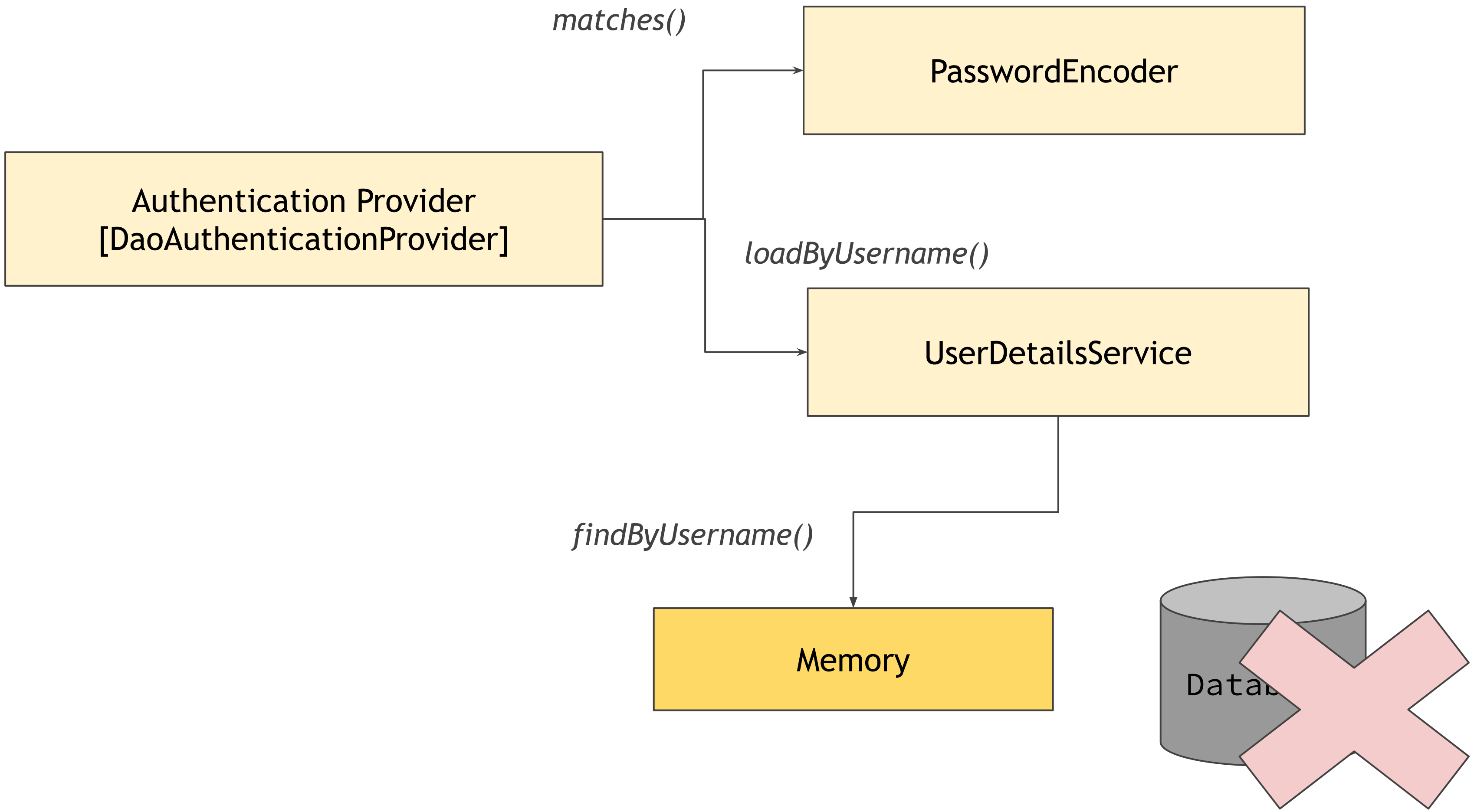
→ *PreAuthenticatedAuthenticationProvider*

→ *OAuth2AuthenticationProvider*

In Memory Authentication

Faisal Memon (EmbarkX)

In-Memory Authentication is storing and managing user credentials directly within the application's memory


matches()
PasswordEncoder
Authentication Provider
[DaoAuthenticationProvider]
loadByUsername()
UserDetailsService
findByUsername()
Memory
Datab

Use Cases

→ *Development and Testing*

→ *Small Applications*

→ *Prototyping*

Benefits

→ *Simplicity*

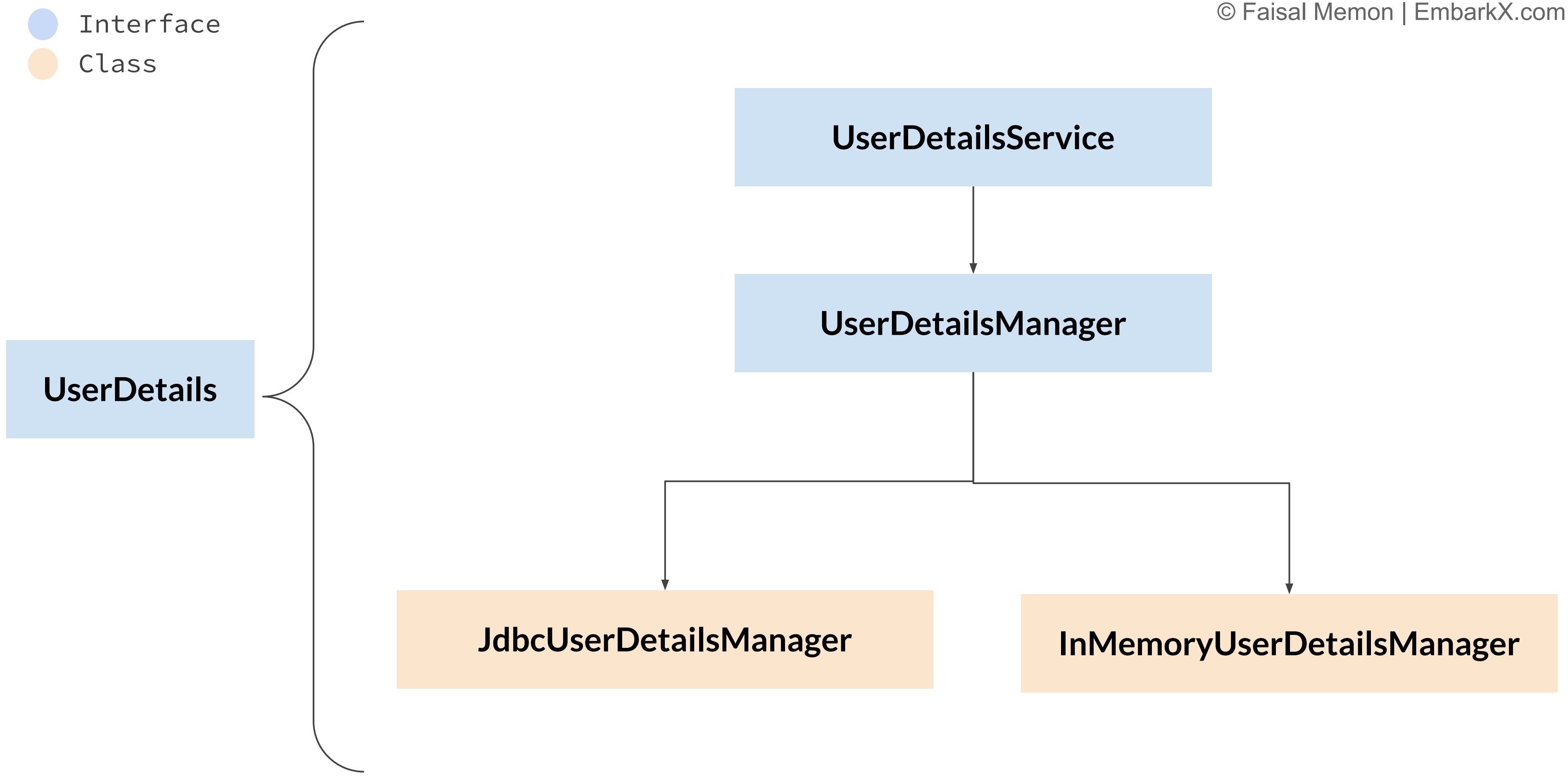
→ *Speed*

→ *Convenience*

Thank you

Core Classes & Interfaces for user management

Faisal Memon (EmbarkX)


Interface
Class
UserDetails
UserDetailsService
UserDetailsManager
JdbcUserDetailsManager
InMemoryUserDetailsManager

UserDetails

→ *The UserDetails interface is a core component in Spring Security that represents a user in the application*

→ *It provides necessary information about the user, such as username, password, and authorities (roles)*

```
String getUsername()
String getPassword()
Collection<? extends GrantedAuthority> getAuthorities()
boolean isAccountNonExpired()
boolean isAccountNonLocked()
boolean isCredentialsNonExpired()
boolean isEnabled()
```

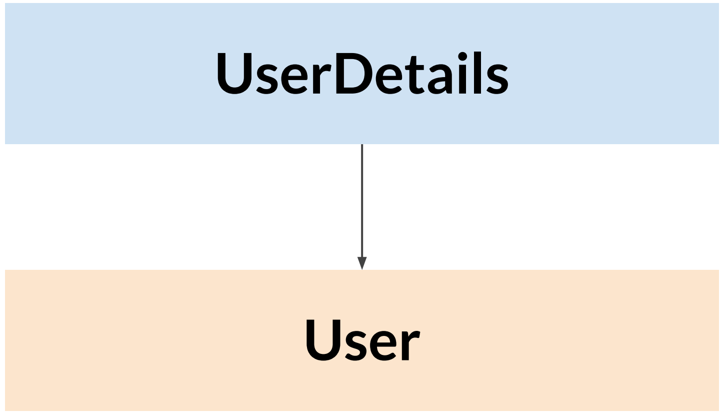
User

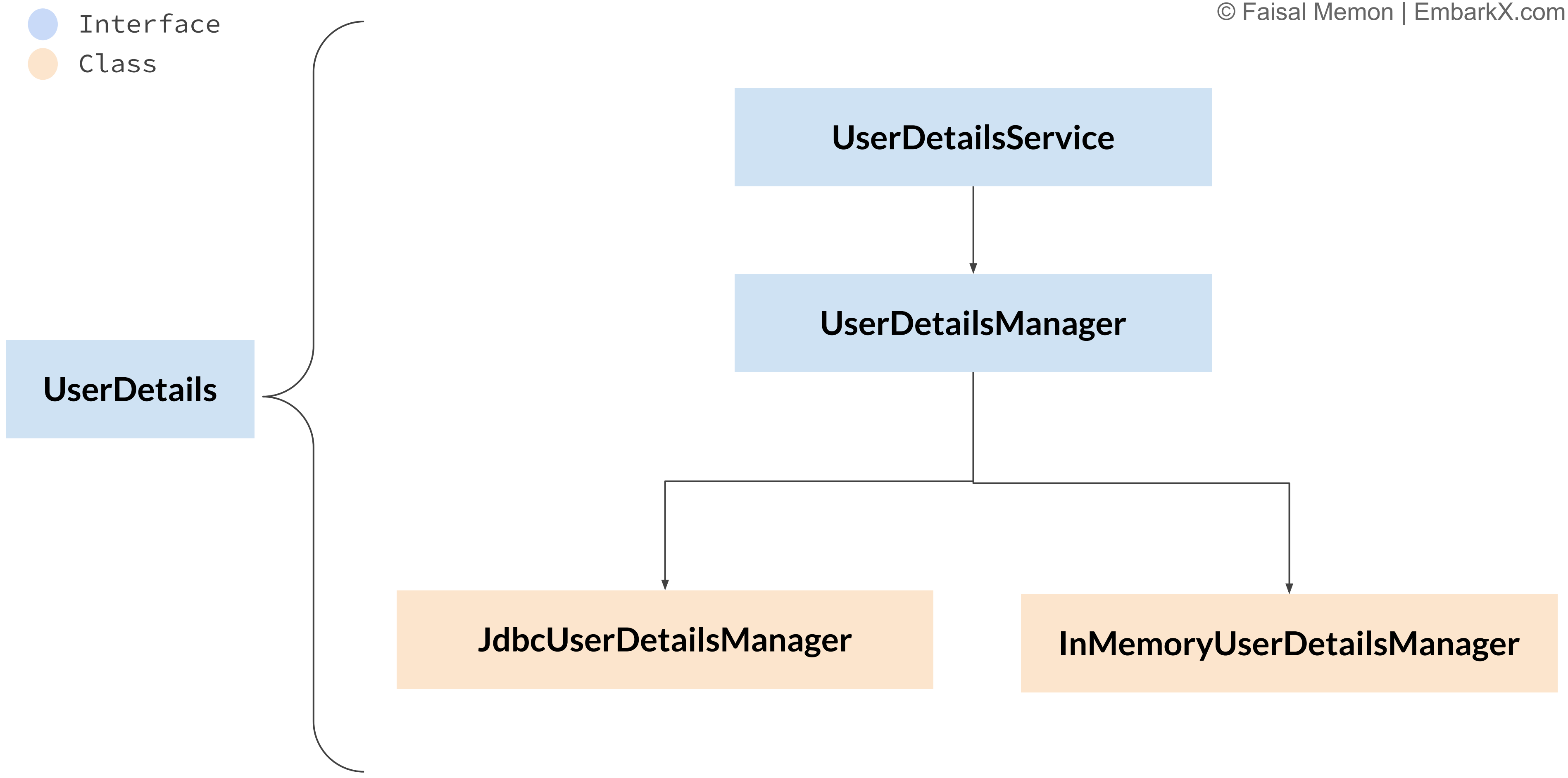
→ *User is a concrete implementation of the UserDetails interface provided by Spring Security.*

→ *It is often used to create a UserDetails object with predefined username, password, and authorities.*

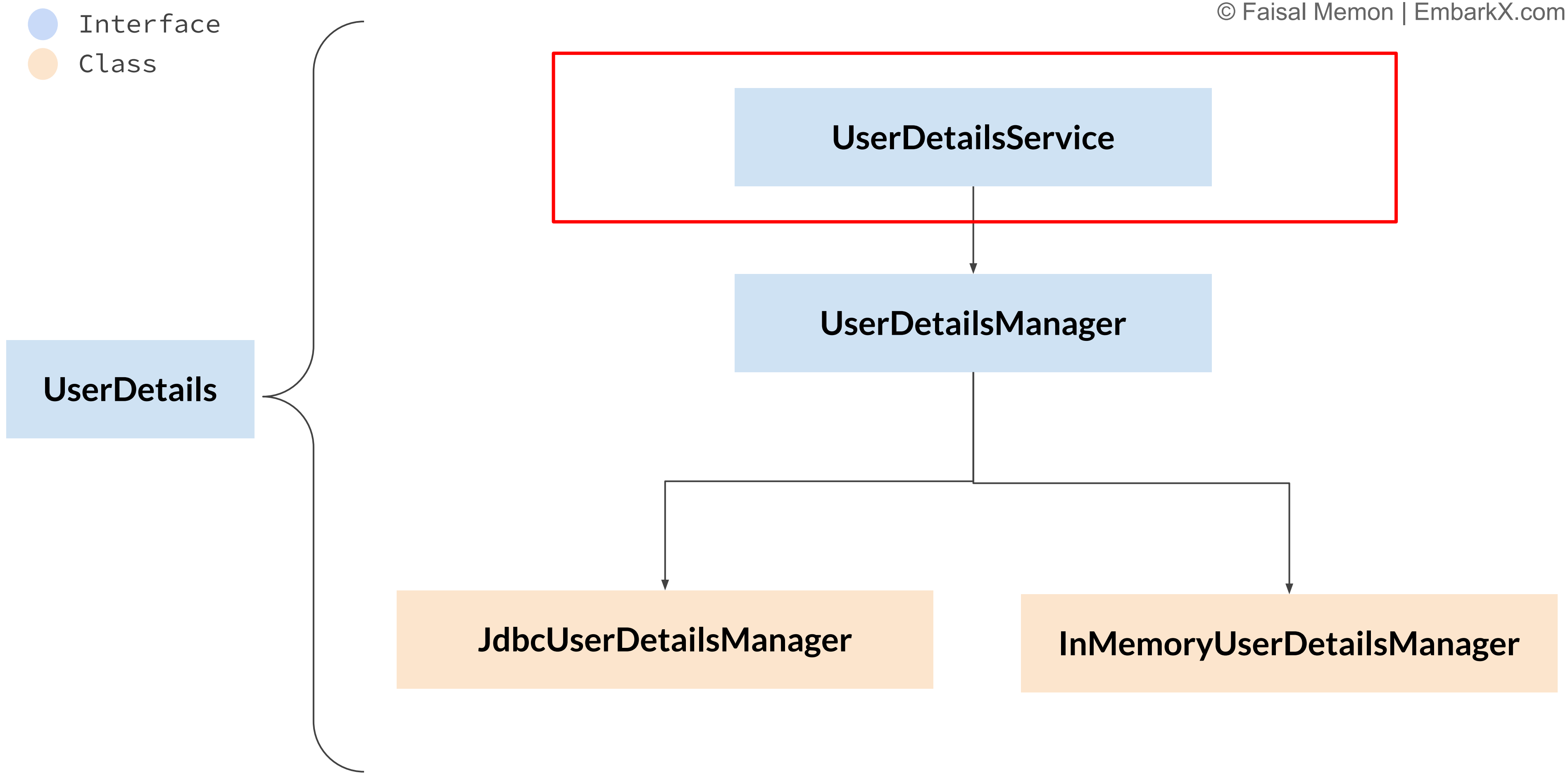
```
UserDetails user = User.withUsername("user")
                       .password("{noop}password")
                       .authorities("ROLE_USER")
                       .build();
```

● Interface
● Class





● Interface
● Class
UserDetails
UserDetailsService
UserDetailsManager
JdbcUserDetailsManager
InMemoryUserDetailsManager



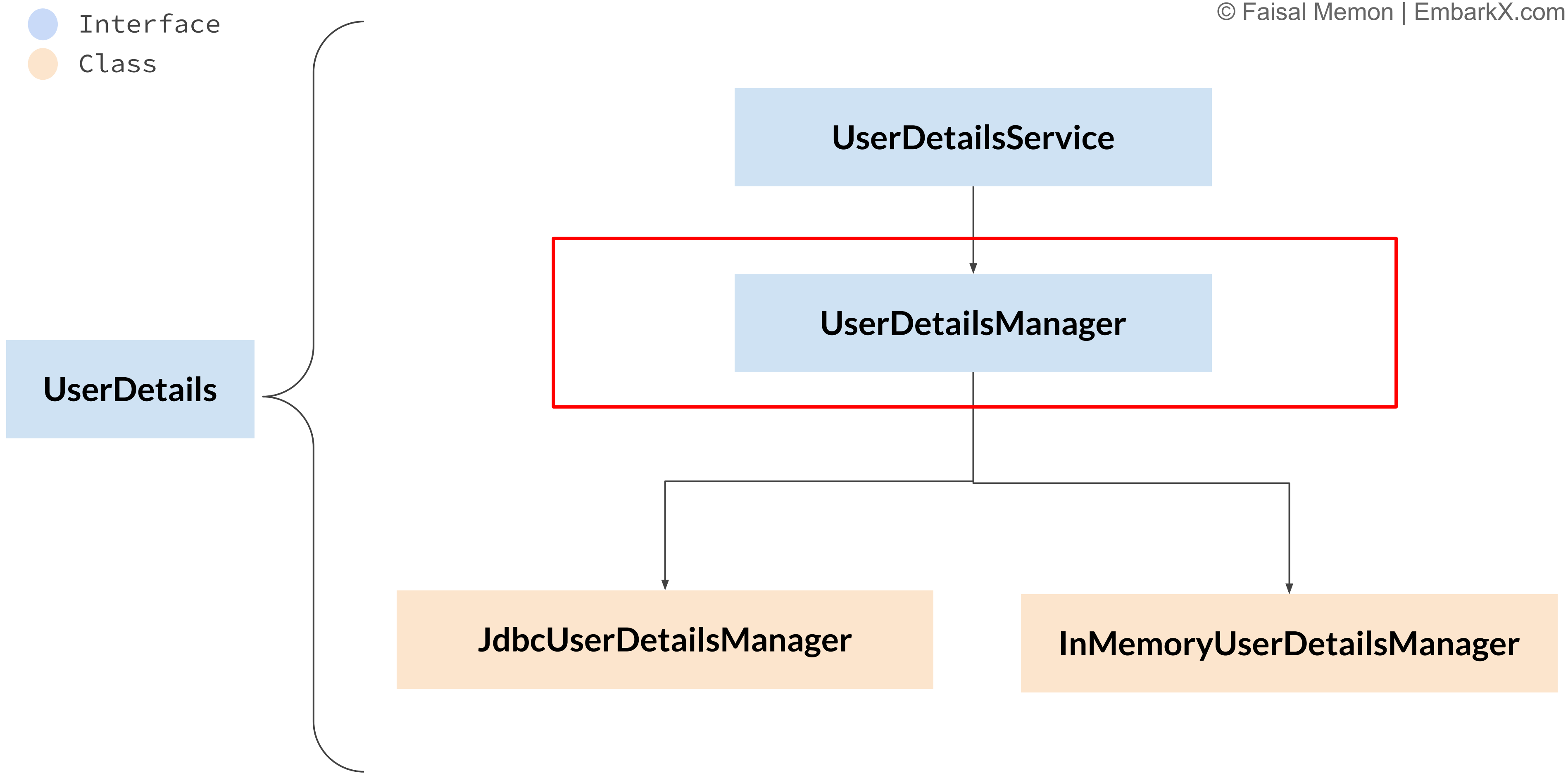
Interface
Class
UserDetails
UserDetailsService
UserDetailsManager
JdbcUserDetailsManager
InMemoryUserDetailsManager

UserDetailsService

→ *The UserDetailsService interface is responsible for retrieving user-related data.*

→ *It has a single method that loads a user based on the username and returns a UserDetails object.*

```
UserDetails loadUserByUsername(String username)
       throws UsernameNotFoundException
```



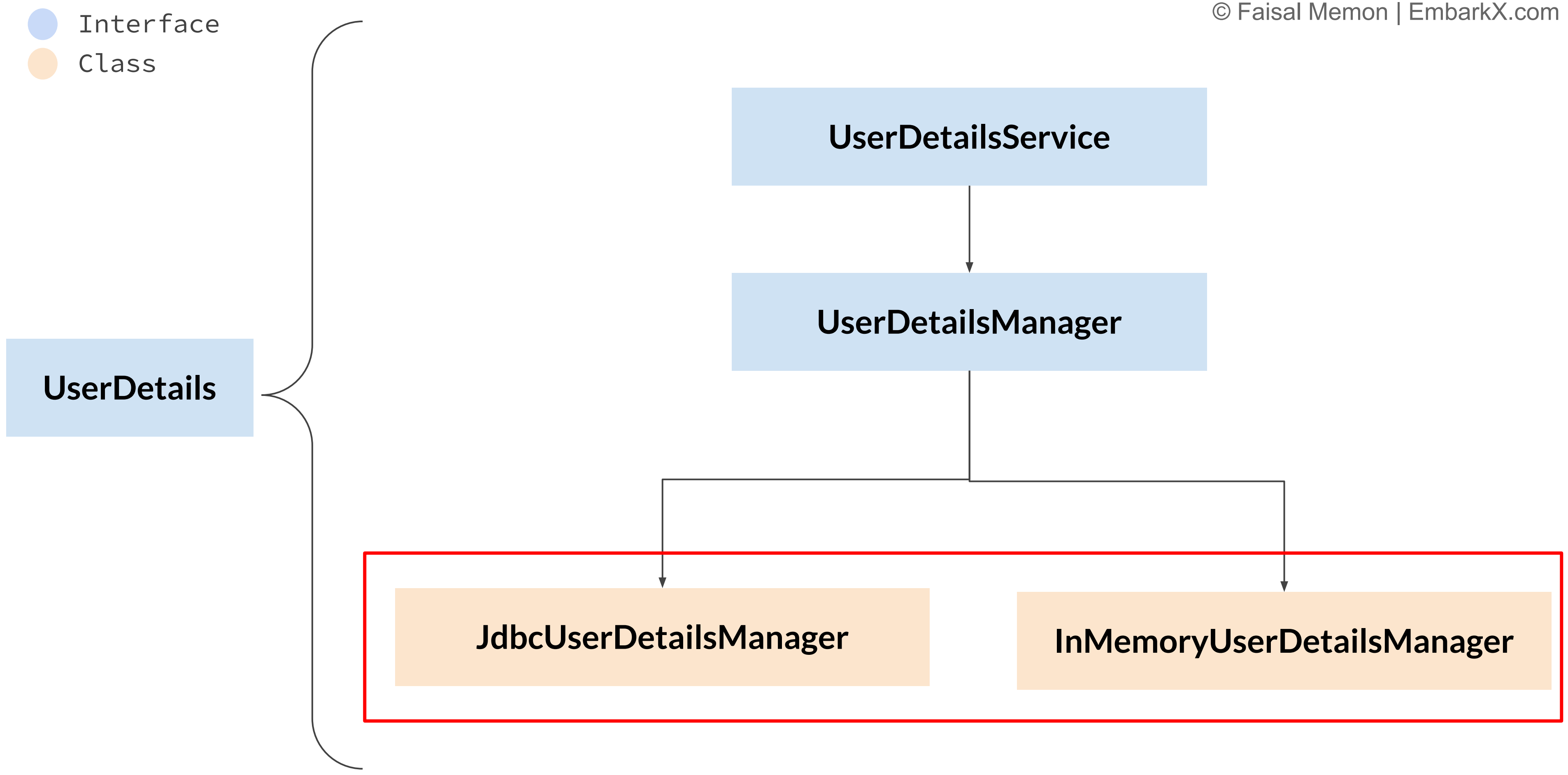
Interface
Class
UserDetails
UserDetailsService
UserDetailsManager
JdbcUserDetailsManager
InMemoryUserDetailsManager

UserDetailsManager

→ *The UserDetailsManager interface in Spring Security extends UserDetailsService and provides additional methods for managing user accounts.*

→ *Provides additional capabilities for managing user accounts, such as creating, updating, and deleting users, as well as changing passwords and checking for user existence.*

```
void createUser(UserDetails user)
void updateUser(UserDetails user)
void deleteUser(String username)
void changePassword(String oldPassword, String newPassword)
boolean userExists(String username)
```


Interface
Class
UserDetails
UserDetailsService
UserDetailsManager
JdbcUserDetailsManager
InMemoryUserDetailsManager

JdbcUserDetailsManager

→ *JdbcUserDetailsManager is a Spring Security implementation of the UserDetailsManager interface that manages user details using a JDBC-based data source*

→ *It provides methods to create, update, delete, and query user accounts, and it interacts with the database using SQL queries.*

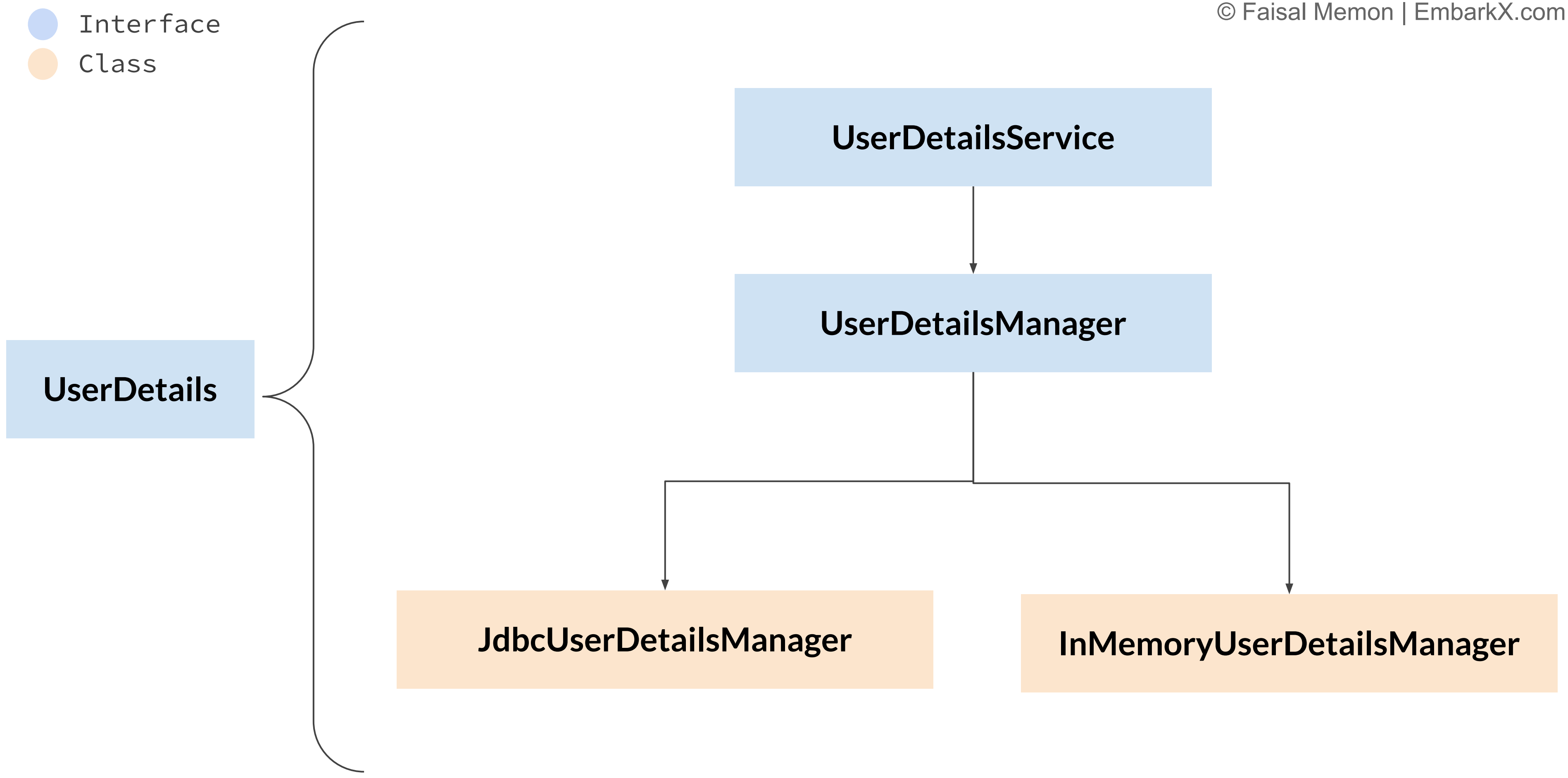
```
void createUser(UserDetails user)
void updateUser(UserDetails user)
void deleteUser(String username)
void changePassword(String oldPassword, String newPassword)
boolean userExists(String username)
UserDetails loadUserByUsername(String username)
```

InMemoryUserDetailsManager

→ *InMemoryUserDetailsManager is another implementation of the UserDetailsManager interface provided by Spring Security.*

→ *It manages user details entirely in memory, which means the user data is stored in memory (RAM) and is not persistent across application restarts.*

```
void createUser(UserDetails user)
void updateUser(UserDetails user)
void deleteUser(String username)
void changePassword(String oldPassword, String newPassword)
boolean userExists(String username)
UserDetails loadUserByUsername(String username)
```


● Interface
● Class
UserDetails
UserDetailsService
UserDetailsManager
JdbcUserDetailsManager
InMemoryUserDetailsManager

Custom User Model

Faisal Memon (EmbarkX)

Why Custom User Model

→ *Extended User Information*

→ *Domain-Specific Requirements*

→ *Custom Authentication and Authorization Logic*

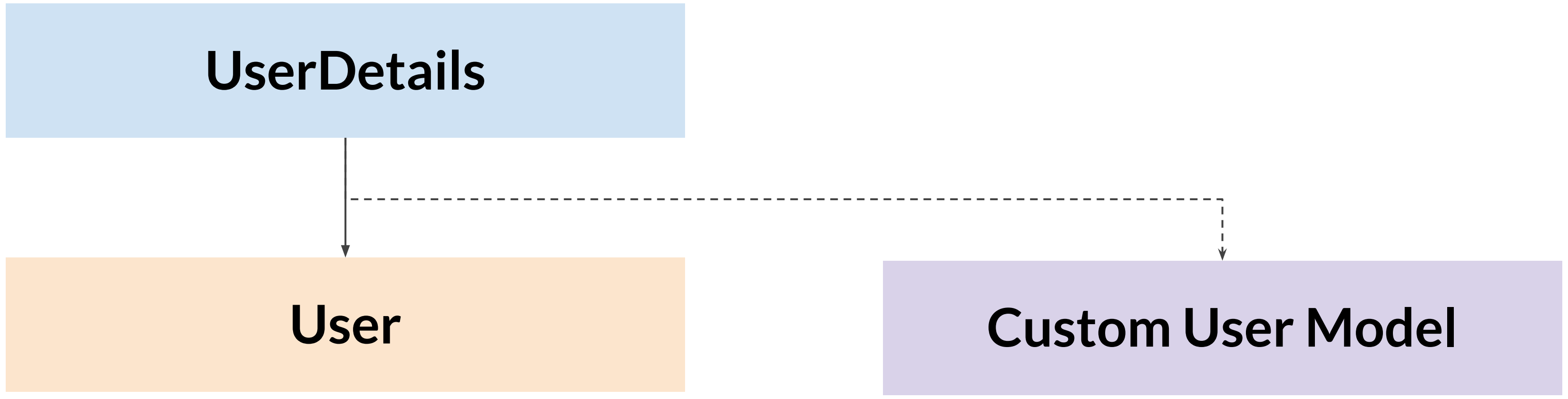
→ *Integration with Other Systems*

→ *Enhanced Security*

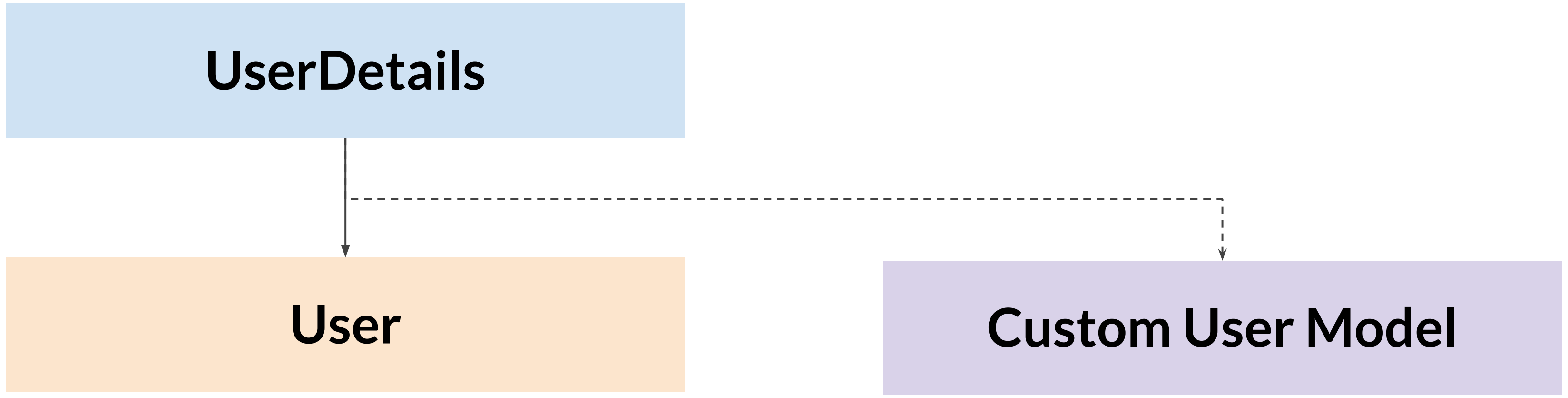
Benefits of a Custom User Model

- *Flexibility*
- *Better Code Organization*
- *Easier Testing*
- *Improved User Experience*

● Interface
● Class



● Interface
● Class



Thank you

Role Based Authorization

Faisal Memon (EmbarkX)

Role-based authorization is a method of restricting access to resources based on the roles assigned to users

Online Banking System

Customer

Role: Can view account balance, transfer money, pay bills

Permissions: Access to personal account details, perform transactions.

Teller

Role: Can manage customer accounts, view transaction history, approve loans

Permissions: Access to customer information, modify accounts, approve transactions.

Admin

Role :Can create or delete user accounts, manage roles, oversee system operations

Permissions: Full system access, user and role management.

Importance

→ *Security*

→ *Manageability*

→ *Scalability*

→ *Flexibility*

API's we need

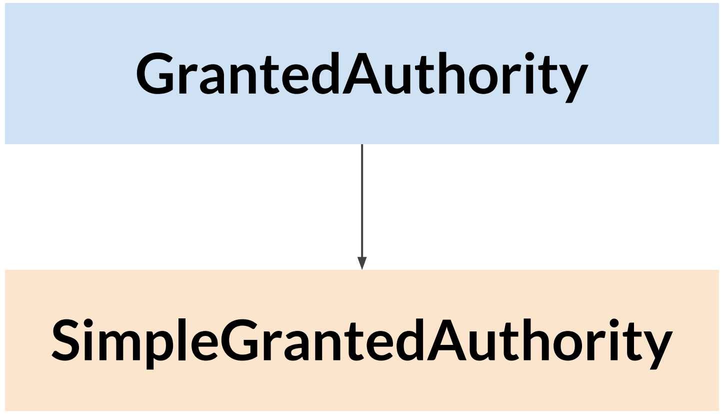
HTTP Method	Endpoint	Description	Parameters	Response
GET	/getusers	Retrieve all users	None	List of User objects (HTTP 200)
PUT	/update-role	Update a user's role	userId (Long, required), roleName (String, required)	Success message (HTTP 200)
GET	/user/{id}	Retrieve a user by ID	id (Path variable, Long, required)	UserDTO object (HTTP 200)

Thank you

Inbuilt classes and interfaces For Authorization

Faisal Memon (EmbarkX)

● Interface
● Class



Managing Access with Annotations

Faisal Memon (EmbarkX)

Spring Security provides **annotations** to secure methods in your services or controllers.

Method-Level Security

→ @PreAuthorize

→ @Secured

→ @RolesAllowed

→ @PostAuthorize

→ @PreFilter and @PostFilter

Example

```
import
org.springframework.security.access.prepost.PreAuthorize;
import org.springframework.stereotype.Service;

@Service
public class AdminService {

    @PreAuthorize("hasRole('ADMIN')")
    public void performAdminTask() {
        // Code for admin task
    }
}
```

Thank you

Restricting Admin Actions

Faisal Memon (EmbarkX)

Techniques

→ *URL-Based Restrictions*

→ *Method-Level Security*

Method Level Security

Faisal Memon (EmbarkX)

Method-level security in Spring Security allows you to apply security constraints directly on methods within your services or controllers

Method-Level Security

→ *@PreAuthorize*

→ *@Secured*

→ *@RolesAllowed*

→ *@PostAuthorize*

→ *@PreFilter and @PostFilter*

@PreAuthorize

Checks the given expression before entering the method

Example

```
@Service
public class DocumentService {

    @PreAuthorize("hasRole('ADMIN')")
    public void deleteDocument(Long documentId) {
        // Method implementation
    }

    @PreAuthorize("hasRole('USER') or hasRole('ADMIN')")
    public Document getDocument(Long documentId) {
        // Method implementation
    }

    @PreAuthorize("#document.owner == authentication.name")
    public void updateDocument(Document document) {
        // Method implementation
    }
}
```

Example

```
@Service
public class DocumentService {

    @PreAuthorize("hasRole('ADMIN')")
    public void deleteDocument(Long documentId) {
        // Method implementation
    }

    @PreAuthorize("hasRole('USER') or hasRole('ADMIN')")
    public Document getDocument(Long documentId) {
        // Method implementation
    }

    @PreAuthorize("#document.owner == authentication.name")
    public void updateDocument(Document document) {
        // Method implementation
    }
}
```

Example

```
@Service
public class DocumentService {

    @PreAuthorize("hasRole('ADMIN')")
    public void deleteDocument(Long documentId) {
        // Method implementation
    }

    @PreAuthorize("hasRole('USER') or hasRole('ADMIN')")
    public Document getDocument(Long documentId) {
        // Method implementation
    }

    @PreAuthorize("#document.owner == authentication.name")
    public void updateDocument(Document document) {
        // Method implementation
    }
}
```

Example

```
@Service
public class DocumentService {

    @PreAuthorize("hasRole('ADMIN')")
    public void deleteDocument(Long documentId) {
        // Method implementation
    }

    @PreAuthorize("hasRole('USER') or hasRole('ADMIN')")
    public Document getDocument(Long documentId) {
        // Method implementation
    }

    @PreAuthorize("#document.owner == authentication.name")
    public void updateDocument(Document document) {
        // Method implementation
    }
}
```

Example

```java
@Service
public class DocumentService {

    @PreAuthorize("hasRole('ADMIN')")
    public void deleteDocument(Long documentId) {
        // Method implementation
    }

    @PreAuthorize("hasRole('USER') or hasRole('ADMIN')")
    public Document getDocument(Long documentId) {
        // Method implementation
    }

    @PreAuthorize("#document.owner == authentication.name")
    public void updateDocument(Document document) {
        // Method implementation
    }
}
```

<u>***@Secured***</u>

Simpler alternative to @PreAuthorize, used to specify roles directly

Example

```java
import org.springframework.security.access.annotation.Secured;
import org.springframework.stereotype.Service;

@Service
public class AccountService {

    @Secured("ROLE_ADMIN")
    public void createAccount(Account account) {
        // Method implementation
    }

    @Secured({"ROLE_USER", "ROLE_ADMIN"})
    public Account getAccount(Long accountId) {
        // Method implementation
    }
}
```

@RolesAllowed

Specifies roles allowed to invoke the method

Example

```java
import javax.annotation.security.RolesAllowed;
import org.springframework.stereotype.Service;

@Service
public class OrderService {

    @RolesAllowed("ROLE_MANAGER")
    public void processOrder(Order order) {
        // Method implementation
    }

    @RolesAllowed({"ROLE_USER", "ROLE_MANAGER"})
    public Order getOrder(Long orderId) {
        // Method implementation
    }
}
```

@PostAuthorize

Checks the given expression after the method has been invoked

Example

```java
import org.springframework.security.access.prepost.PostAuthorize;
import org.springframework.stereotype.Service;

@Service
public class ReportService {

    @PostAuthorize("returnObject.owner == authentication.name")
    public Report getReport(Long reportId) {
        // Method implementation
        return report;
    }
}
```

@PreFilter and @PostFilter

The @PreFilter and @PostFilter annotations filter collections or arrays passed as method arguments or returned by the method.

Example

```
import org.springframework.security.access.prepost.PreFilter;
import org.springframework.security.access.prepost.PostFilter;
import org.springframework.stereotype.Service;

@Service
public class MessageService {

    @PreFilter("filterObject.owner == authentication.name")
    public void sendMessages(List<Message> messages) {
        // Method implementation
    }

    @PostFilter("filterObject.owner == authentication.name")
    public List<Message> getMessages() {
        // Method implementation
        return messages;
    }
}
```

Scenarios

Admin-Only Actions

Methods that should only be accessible to administrators can be secured using @PreAuthorize or @Secured annotations with the ADMIN role.

Role-Based Access Control

Methods accessible by multiple roles can be defined using @PreAuthorize with OR conditions or @Secured with multiple roles.

Ownership and Contextual Access

Methods that require checks on ownership or other contextual conditions can use @PreAuthorize with SpEL expressions to enforce these rules.

Scenarios

Post-Invocation Security

Methods that return data requiring post-invocation security checks can use @PostAuthorize to enforce access control on the returned object.

Filtering Collections

Methods dealing with collections of objects can use @PreFilter and @PostFilter to filter the collection based on security constraints.

Thank you

URL Based Restrictions

Faisal Memon (EmbarkX)

Spring Security allows you to configure URL-based restrictions in your security configuration.

Example

```java
@Configuration
@EnableWebSecurity
public class SecurityConfig {

    @Bean
    public SecurityFilterChain filterChain(HttpSecurity http) throws Exception {
        http
            .authorizeHttpRequests(auth -> auth
                .requestMatchers("/api/auth/**").permitAll()
                .requestMatchers("/admin/**").hasRole("ADMIN")
                .requestMatchers("/images/**").permitAll()
                .anyRequest().authenticated())
            .httpBasic();

        return http.build();
    }
}
```

Example

```
@Configuration
@EnableWebSecurity
public class SecurityConfig {

    @Bean
    public SecurityFilterChain filterChain(HttpSecurity http) throws Exception {
        http
            .authorizeHttpRequests(auth -> auth
                .requestMatchers("/api/auth/**").permitAll()
                .requestMatchers("/admin/**").hasRole("ADMIN")
                .requestMatchers("/images/**").permitAll()
                .anyRequest().authenticated())
            .httpBasic();

        return http.build();
    }
}
```

Example

```
@Configuration
@EnableWebSecurity
public class SecurityConfig {

    @Bean
    public SecurityFilterChain filterChain(HttpSecurity http) throws Exception {
        http
            .authorizeHttpRequests(auth -> auth
                .requestMatchers("/api/auth/**").permitAll()
                .requestMatchers("/admin/**").hasRole("ADMIN")
                .requestMatchers("/images/**").permitAll()
                .anyRequest().authenticated())
            .httpBasic();

        return http.build();
    }
}
```

Example

```
@Configuration
@EnableWebSecurity
public class SecurityConfig {

    @Bean
    public SecurityFilterChain filterChain(HttpSecurity http) throws Exception {
        http
            .authorizeHttpRequests(auth -> auth
                .requestMatchers("/api/auth/**").permitAll()
                .requestMatchers("/admin/**").hasRole("ADMIN")
                .requestMatchers("/images/**").permitAll()
                .anyRequest().authenticated())
            .httpBasic();

        return http.build();
    }
}
```

Example

```
@Configuration
@EnableWebSecurity
public class SecurityConfig {

    @Bean
    public SecurityFilterChain filterChain(HttpSecurity http) throws Exception {
        http
            .authorizeHttpRequests(auth -> auth
                .requestMatchers("/api/auth/**").permitAll()
                .requestMatchers("/admin/**").hasRole("ADMIN")
                .requestMatchers("/images/**").permitAll()
                .anyRequest().authenticated())
            .httpBasic();

        return http.build();
    }
}
```

Method Level Security vs RequestMatchers

Faisal Memon (EmbarkX)

	Method Level Security	RequestMatchers Approach
Definition	Uses annotations to secure individual methods.	Configures security based on URL patterns in HTTP configuration.
Key Annotations/Methods	`@PreAuthorize`, `@PostAuthorize`, `@Secured`, `@RolesAllowed`	`requestMatchers`
Granularity	Fine-grained control over individual methods.	Coarse-grained control based on URL patterns.
Configuration Location	Annotations on methods or classes in service or controller layers.	Centralized in security configuration file.
Flexibility	Highly flexible with complex expressions using SpEL.	Clear and straightforward URL-based rules.
Impact on Business Logic	Directly couples security with business logic.	Keeps security rules separate from business logic.
Management Complexity	Can be verbose; requires annotations on each method.	Easier to manage with all rules in one configuration file.
Use Case Suitability	Ideal for detailed control and complex conditions.	Ideal for simple and maintainable URL-based security.

	Method Level Security	**RequestMatchers Approach**
Examples	@PreAuthorize("hasRole('ROLE_ADMIN')") @Secured("ROLE_ADMIN")	.requestMatchers("/admin/**").hasRole("ADMIN")
Best Use Cases	- Securing service methods accessed by various controllers. - Applying role-based access with additional conditions.	- Securing web applications with clear URL patterns for different roles. - Enforcing access control on REST APIs based on URL structures.
Pros	- Provides detailed access control. - Can apply complex security logic. - Ensures security at the business logic layer.	- Centralized management. - Clear URL-based rules. - Less intrusive to business logic.
Cons	- Requires annotations on each secured method. - Tightly coupled with business logic.	- Less granularity compared to method-level security. - Potential for overlapping or conflicting rules with method-level security.

Scenario	Preferred Approach	Reasoning
Need for Fine-Grained Control	Method Level Security	Provides detailed control over individual methods.
Complex Security Logic	Method Level Security	Can apply complex conditions using SpEL.
Service Layer Security	Method Level Security	Ensures security checks directly at the service layer.
Centralized Security Management	RequestMatchers Approach	Centralized configuration in one place.
Clear URL-Based Security	RequestMatchers Approach	Simple and clear rules based on URL patterns.
Simplicity and Maintainability	RequestMatchers Approach	Easier to manage without modifying business logic.

Combining Both Approaches

Thank You

Password Security and Password Encoding

Faisal Memon (EmbarkX)

Password security refers to the measures and practices used to protect passwords from being stolen, guessed, or otherwise compromised

Why is Password Security Important?

→ *Protection of Sensitive Data*

→ *Prevent Unauthorized Access*

→ *Compliance and Legal Requirements*

→ *Maintaining Trust*

Password encoding is the process of transforming a password into a different format using an algorithm.

Introduction to Custom Filters

Faisal Memon (EmbarkX)

Spring Security filters are Java components that intercept HTTP requests and responses

Custom Security Filter Scenarios

Custom Authentication and Authorization

Custom filters can implement specialized authentication logic, such as token-based authentication, custom headers, or multi-factor authentication (MFA)

Custom Security Filter Scenarios

Rate Limiting

To prevent abuse of API endpoints, custom filters can implement rate limiting logic

IP Whitelisting and Blacklisting

Restrict access of API endpoints to certain IP addresses

Geo-Blocking

Restrict access of API endpoints to certain locations

Custom Security Filter Scenarios

Compliance and Logging

Custom filters can be used to implement detailed logging mechanisms for compliance purposes

Integration with External Systems

Custom filters can integrate with external systems or third-party services

Handling Cross-Cutting Concerns

Custom filters can handle cross-cutting concerns such as logging, transaction management, or modifying requests and responses

Default Filter Chain

Faisal Memon (EmbarkX)

```
DisableEncodeUrlFilter
WebAsyncManagerIntegrationFilter
SecurityContextHolderFilter
HeaderWriterFilter
CorsFilter
CsrfFilter
LoggingFilter
LogoutFilter
OAuth2AuthorizationRequestRedirectFilter
OAuth2LoginAuthenticationFilter
AuthTokenFilter
RequestCacheAwareFilter
SecurityContextHolderAwareRequestFilter
AnonymousAuthenticationFilter
ExceptionTranslationFilter
AuthorizationFilter
```

Filter Lifecycle

Faisal Memon (EmbarkX)

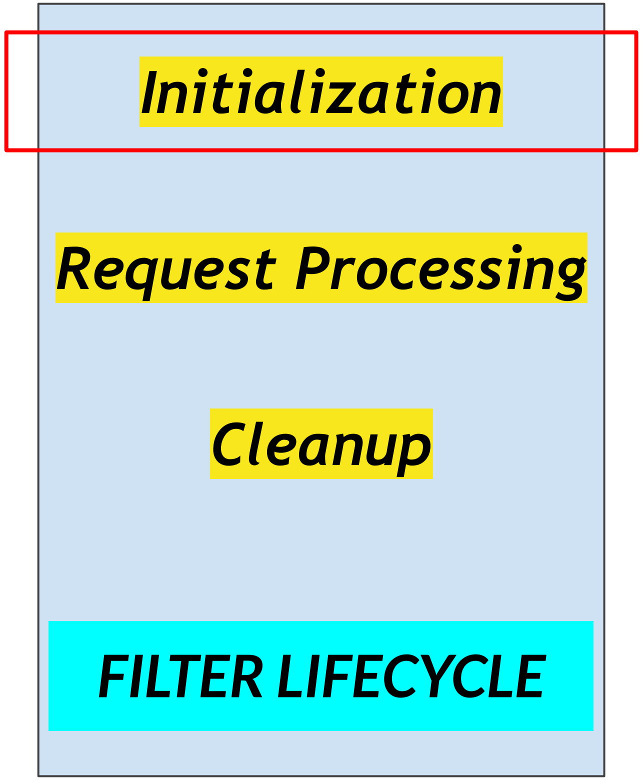
Filters are part of the Servlet API and have a well-defined lifecycle managed by the Servlet container.

Initialization

Request Processing

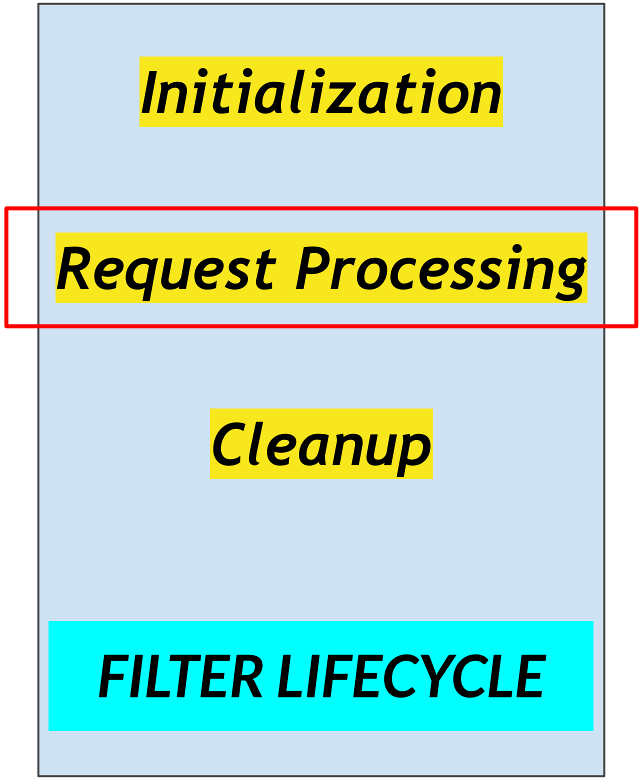
Cleanup

FILTER LIFECYCLE



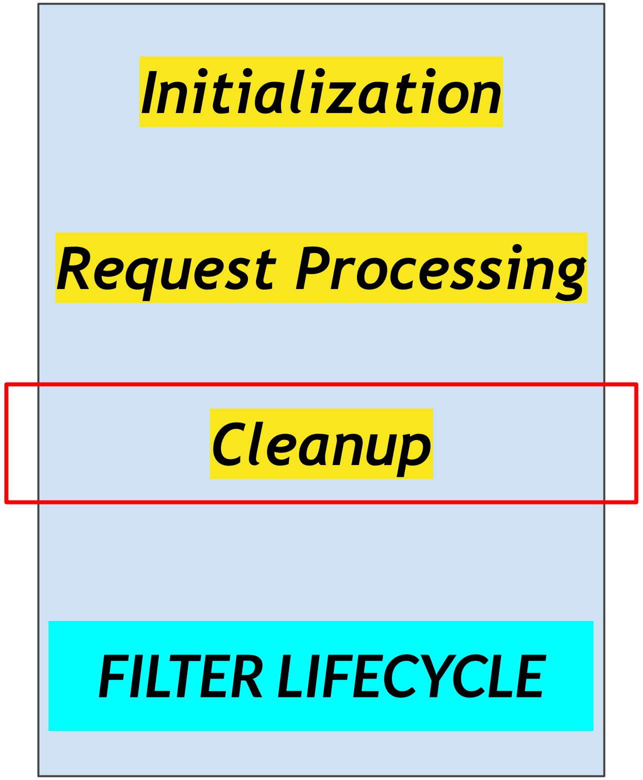
→ **init**: This method is called once when the filter is first created.

→It is used to perform any necessary setup or resource allocation.



→ **doFilter**: This method is called every time a request/response pair is passed through the filter chain.

→It performs the main filtering task



→ **destroy**: This method is called once when the filter is being removed from service.

→It is used to release any resources allocated

Where do these methods exist

init(FilterConfig filterConfig)

This method exists in the Filter interface and is called by the Servlet container

doFilter(ServletRequest request, ServletResponse response, FilterChain chain)

This method also exists in the Filter interface and is called by the Servlet container for each request/response pair that passes through the filter

destroy()

This method is part of the Filter interface and is called by the Servlet container when the filter is being taken out of service

Inbuilt Classes for Filter Implementation

Faisal Memon (EmbarkX)

Inbuilt Classes for Filter Implementation

- *Using OncePerRequestFilter*
- *Using GenericFilterBean*

OncePerRequestFilter

→ *OncePerRequestFilter is an abstract base class provided by Spring Security.*

→ *Ensures that the filter is executed only once per request.*

→ *Simplifies filter implementation by handling repeated invocations.*

GenericFilterBean

→ *GenericFilterBean is a base class provided by Spring Framework.*

→ *It provides a simpler way to create filters without directly implementing the Filter interface.*

→ *Requires implementing the doFilter method.*

Adding Filters into FilterChain

→ *addFilterBefore(Filter filter, Class<? extends Filter> beforeFilter)*

→ *addFilterAfter(Filter filter, Class<? extends Filter> afterFilter)*

Advanced Custom Filter Scenarios

Faisal Memon (EmbarkX)

Custom Filter Scenarios

- *Combining Multiple Filters*
- *Conditional Filters Based on Request Attributes*
- *Dynamic Filter Configuration*

Combining Multiple Filters

→ Ensure the correct order of filters to maintain the logical flow and prevent bypassing security measures.

→ Combine filters to handle complex scenarios that require multiple checks or actions.

Conditional Filters

→ *Use request attributes or headers to decide whether to apply certain filters.*

→ *Implement logic within filters to conditionally process requests.*

```
@Component
public class UserAgentFilter extends OncePerRequestFilter {

    @Override
    protected void doFilterInternal(HttpServletRequest request,
HttpServletResponse response, FilterChain filterChain)
            throws ServletException, IOException {
        String userAgent = request.getHeader("User-Agent");
        if (userAgent != null && userAgent.contains("Mozilla")) {
            // Additional processing for requests from browsers
            System.out.println("Request from browser");
        }
        filterChain.doFilter(request, response);
    }
}
```

Dynamic Filter Configuration

→ *Use Spring's configuration properties or externalized configuration to control filter behavior.*

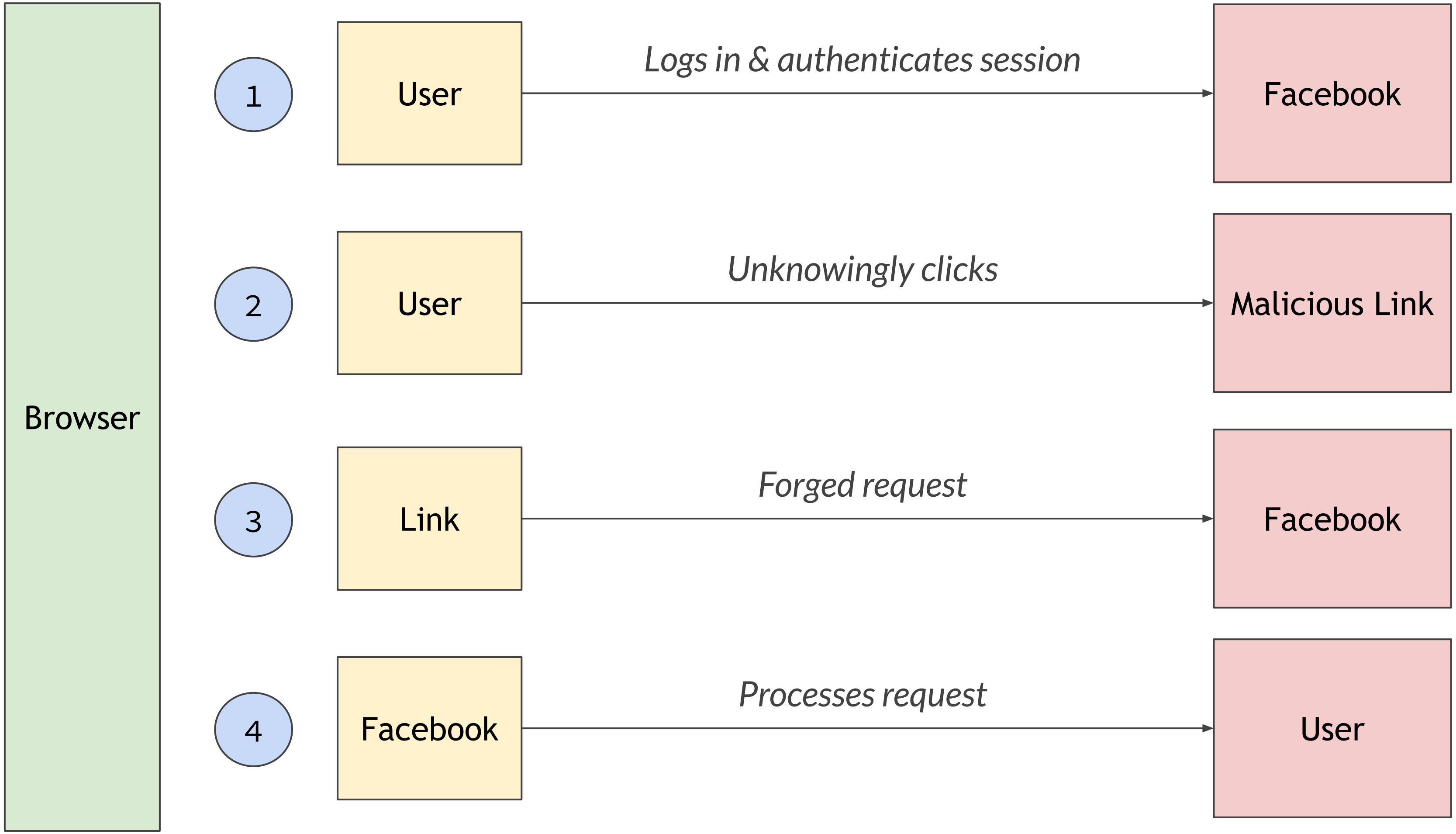
→ *Enable or disable filters based on runtime conditions.*

```java
import org.springframework.beans.factory.annotation.Value;
import org.springframework.stereotype.Component;

@Component
public class DynamicIpWhitelistingFilter extends OncePerRequestFilter {

    @Value("${whitelisted.ips}")
    private List<String> whitelistedIps;

    @Override
    protected void doFilterInternal(HttpServletRequest request, HttpServletResponse
response, FilterChain filterChain)
            throws ServletException, IOException {
        String clientIp = request.getRemoteAddr();
        if (!whitelistedIps.contains(clientIp)) {
            response.sendError(HttpServletResponse.SC_FORBIDDEN, "Access Denied");
            return;
        }
        filterChain.doFilter(request, response);
    }
}
```

What is CSRF?

Faisal Memon (EmbarkX)

CSRF (Cross-Site Request Forgery) is a type of malicious exploit of a website where unauthorized commands are transmitted from a user


Browser
1
User
Logs in & authenticates session
Facebook
2
User
Unknowingly clicks
Malicious Link
3
Link
Forged request
Facebook
4
Facebook
Processes request
User

Bank Transfer Attack

The user visits a malicious website while logged in. This site contains an embedded script that sends a POST request to the bank's transfer money endpoint.

```
<!-- Malicious website -->
<img src="http://trustedbank.com/transfer?amount=1000&toAccount=attacker"
style="display:none;" />
```

Changing User Email

The user clicks on a link in an email that directs them to a malicious website. This site sends a request to change the user's email address to the attacker's email.

```html
<!-- Malicious website -->
<form action="http://socialmedia.com/change-email" method="POST">
  <input type="hidden" name="email" value="attacker@example.com" />
  <input type="submit" value="Change Email" />
</form>
```

Subscription Management

A malicious site the user visits sends a request to the service to unsubscribe or change subscription preferences.

```
<!-- Malicious website -->

<img src="http://onlineservice.com/unsubscribe" style="display:none;" />
```

Posting on Forums

The attacker crafts a request to post spam or malicious content on the forum. Example: The user visits a malicious site that sends a request to the forum's post endpoint.

```
<!-- Malicious website -->
<form action="http://forum.com/post" method="POST">
  <input type="hidden" name="content" value="Buy cheap products at
spammywebsite.com" />
  <input type="submit" value="Post" />
</form>
```

E-commerce Purchases

The attacker crafts a request to purchase an item using the user's account.

```html
<!-- Malicious website -->
<form action="http://ecommerce.com/purchase" method="POST">
  <input type="hidden" name="item" value="expensive-gadget" />
  <input type="hidden" name="quantity" value="1" />
  <input type="submit" value="Buy" />
</form>
```

Impact of Successful CSRF Attacks

→ *Financial Loss*

→ *Compromised User Accounts*

→ *Reputation Damage*

→ *Service Disruption*

→ *Data Integrity Issues*

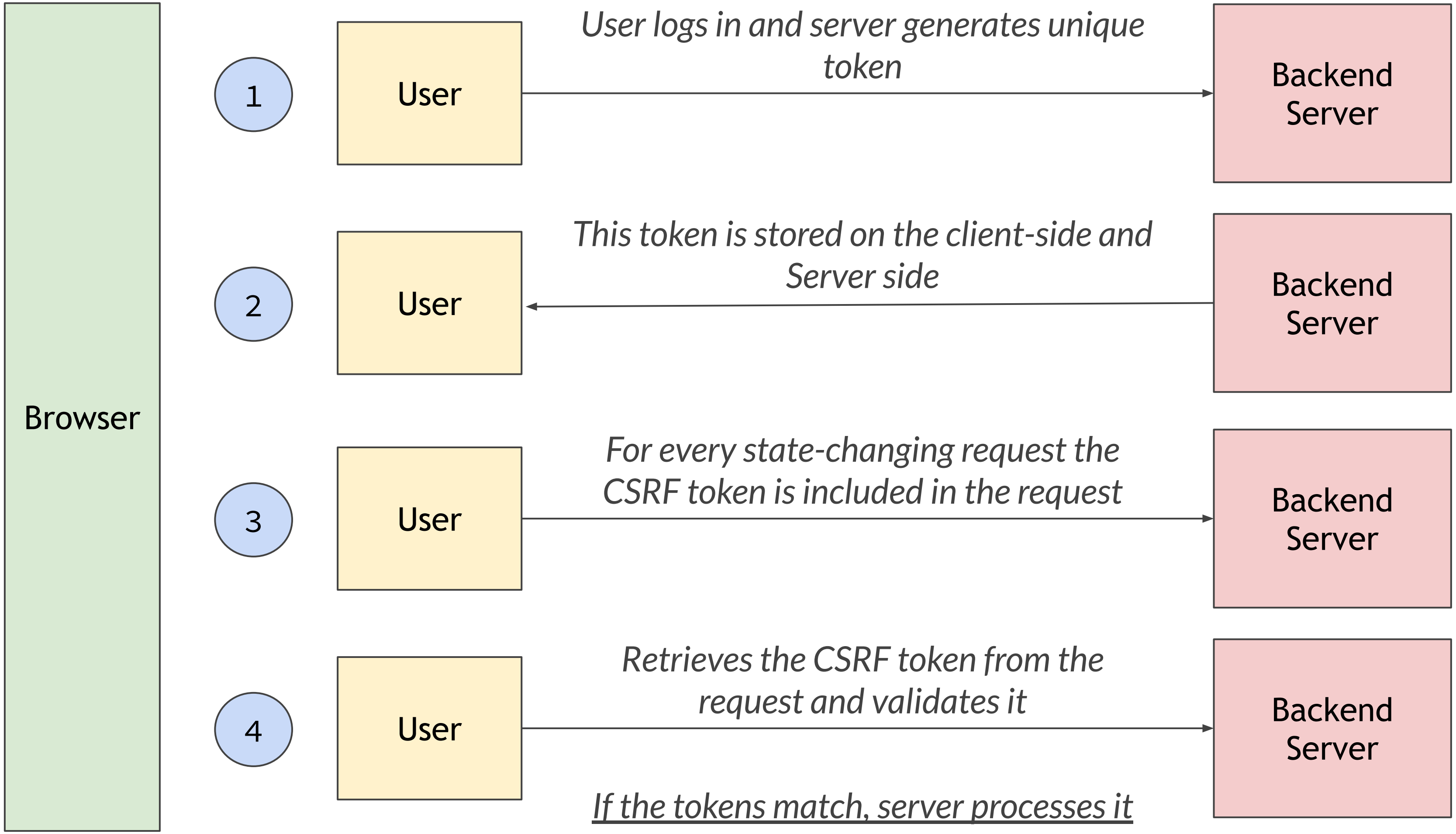
→ *Legal and Compliance Risks*

Thank you

What is CSRF Protection?

Faisal Memon (EmbarkX)

CSRF (Cross-Site Request Forgery) protection is essential to prevent unauthorized actions in web applications when a user is authenticated


Browser
1
User
User logs in and server generates unique token
Backend Server
2
User
This token is stored on the client-side and Server side
Backend Server
3
User
For every state-changing request the CSRF token is included in the request
Backend Server
4
User
Retrieves the CSRF token from the request and validates it
Backend Server
If the tokens match, server processes it

Sign In + Sign Up

Faisal Memon (EmbarkX)

HTTP Method	Path	Description	Request Body	Response Body
POST	/public/signin	Authenticate a user and return a JWT token	LoginRequest	LoginResponse
POST	/public/signup	Register a new user	SignupRequest	MessageResponse
GET	/username	Get the current authenticated username	N/A	String
GET	/user	Get details of the authenticated user	N/A	UserInfoResponse

Auditing and it’s needed

Faisal Memon (EmbarkX)

What is Auditing

Auditing refers to the systematic recording, tracking, and examination of activities and events within the system

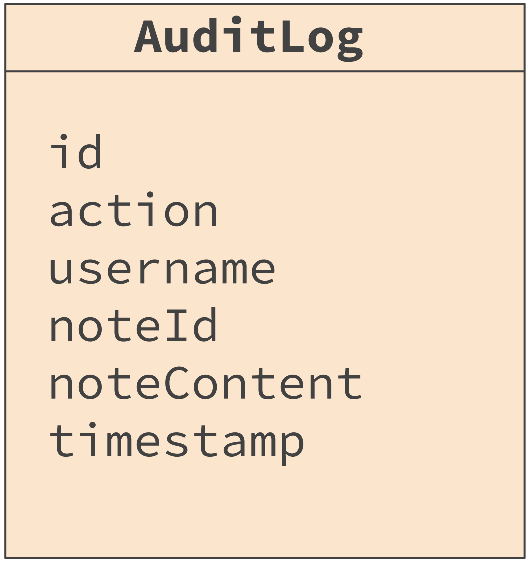
Need

→ *Security*

→ *Compliance*

→ *Accountability*

→ *Operational Efficiency*


AuditLog
id
action
username
noteId
noteContent
timestamp

Endpoint	HTTP Method	Authorization	Description	Parameters	Response
`/api/audit`	`GET`	`ROLE_ADMIN`	Retrieve all audit logs.	None	List of `AuditLog` objects
`/api/audit/note/{id}`	`GET`	`ROLE_ADMIN`	Retrieve audit logs for a specific note by its ID.	`id` (Path Variable, Long)	List of `AuditLog` objects

Admin Actions | Building the Backend

Faisal Memon (EmbarkX)

HTTP Method	Endpoint	Description	Parameters	Response
PUT	/update-lock-status	Updates the account lock status of a user	userId (Long): ID of the user lock (boolean): Lock status	200 OK: "Account lock status updated"
GET	/roles	Retrieves all roles	None	200 OK: List of roles
PUT	/update-expiry-status	Updates the account expiry status of a user	userId (Long): ID of the user expire (boolean): Expiry status	200 OK: "Account expiry status updated"
PUT	/update-enabled-status	Updates the account enabled status of a user	userId (Long): ID of the user enabled (boolean): Enabled status	200 OK: "Account enabled status updated"
PUT	/update-credentials-expiry-status	Updates the credentials expiry status of a user	userId (Long): ID of the user expire (boolean): Credentials expiry status	200 OK: "Credentials expiry status updated"
PUT	/update-password	Updates the password of a user	userId (Long): ID of the user password (String): New password	200 OK: "Password updated" 400 BAD REQUEST: Error message

Password Reset

Faisal Memon (EmbarkX)

Password Reset URL

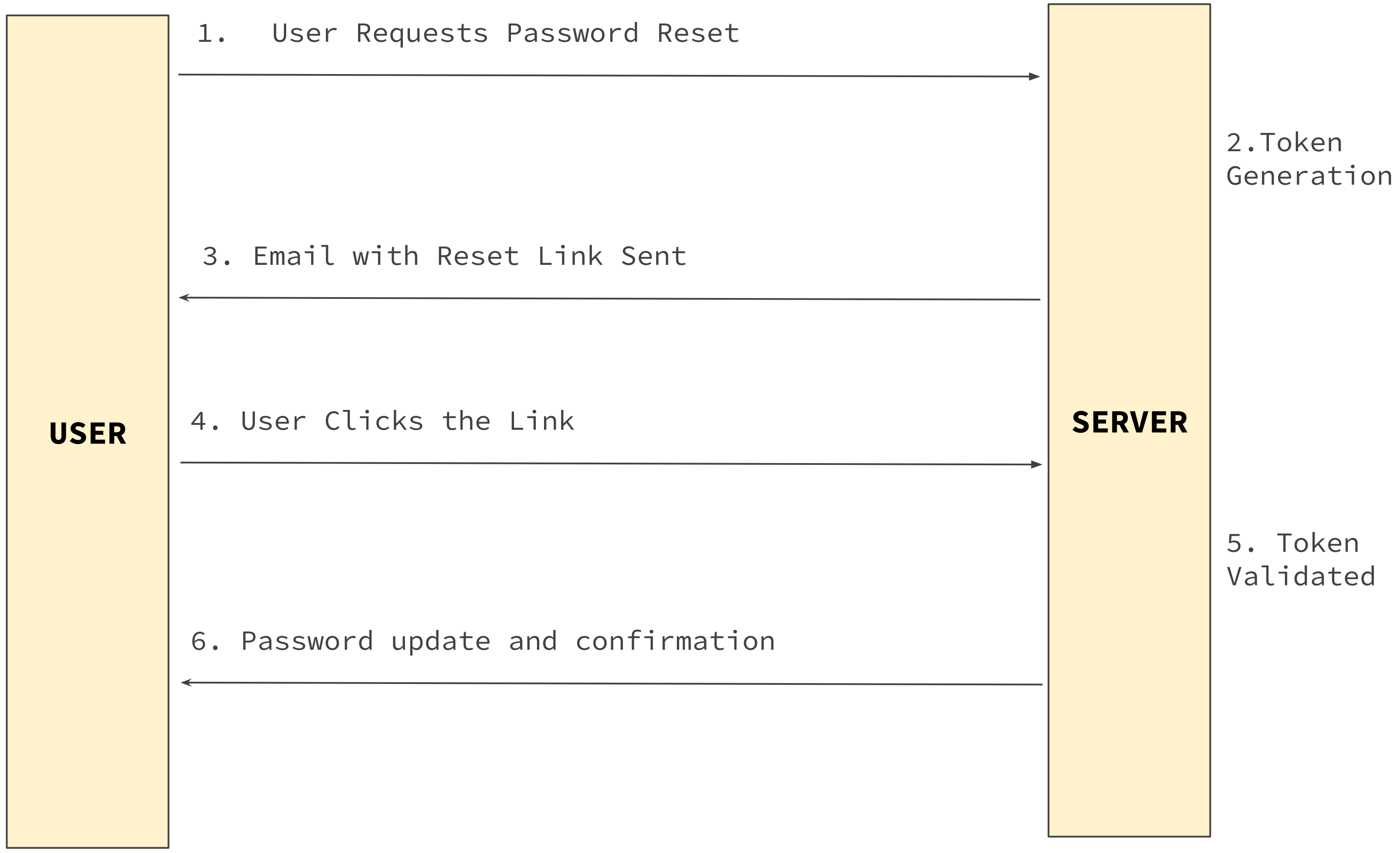
```
https://www.example.com/reset-password?token=123e4567-e89b-12d3-a456-4283
```

Password Reset URL

```
https://www.example.com/reset-password?token=123e4567-e89b-12d3-a456-4283
```

Front end URL

Token


USER
SERVER
1. User Requests Password Reset
2.Token Generation
3. Email with Reset Link Sent
4. User Clicks the Link
5. Token Validated
6. Password update and confirmation

Password Reset Functionality | The Frontend

Faisal Memon (EmbarkX)

Reset Password Flow

→ *The user clicks the link in the reset email and is redirected to the reset password page*

→ *The reset token is extracted from the URL*

→ *The form data (new password) is captured, and front end sends the new password and token to the backend.*

→ *Frontend sends a POST request to the /auth/public/reset-password endpoint with the new password and token*

→ *Password updated and user sees success message*

What do we need?

→ A component which allows users to request a password reset link by submitting their email address [ForgotPassword]

→ A component allows users to reset their password using the token received in the email [ResetPassword]

OAuth2

Faisal Memon (EmbarkX)

You had to **share** your credentials all the time

Problems

→ *Security Risk*

→ *Limited Control*

→ *Inconvenience*

What is OAuth?

OAuth (Open Authorization) is a standard protocol that allows users to grant third-party applications access to their information without sharing their passwords.

Why is OAuth Needed?

OAuth is needed to enable secure and easy access to user information by third-party applications without compromising the user's credentials (like passwords).

OAuth solves the problem of sharing sensitive login credentials directly with third-party applications.

Summary

→ ***What**: OAuth lets apps access your information without needing your password.*

→ ***Why**: It's safer because you don't have to share your password with other apps.*

→ ***Problem Solved**: Before OAuth, apps needed your password to get your info, which was risky.*

→ ***How It Worked Before**: You had to give your password to every app, which was unsafe and inconvenient.*

→ ***How OAuth Works Now**: You log in through a trusted service (like Google), give permission, and the app gets a special token to access your info without needing your password.*

Key Terms

→ ***Resource Owner (User)****: person who owns the account*

→ ***Third-Party Application****: This is the application that wants to access to your account*

→ ***Resource Server****: This is the server that holds data that application wants to access.*

→ ***Authorization Server****: This server handles the authentication (logging in) and authorization (granting permissions)*

→ ***Client****: This is the application that requests access to the resource server on behalf of the user.*

Example

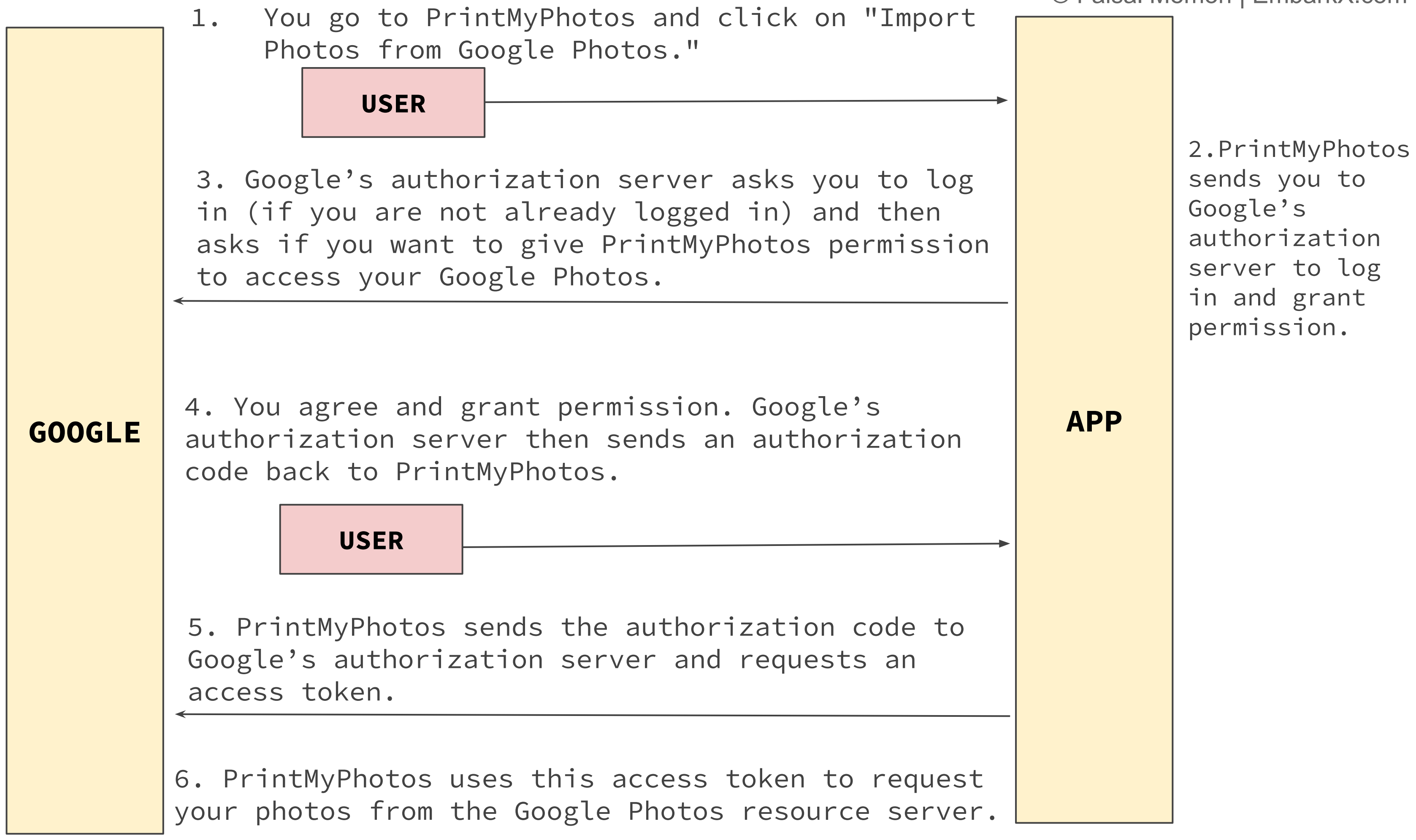
→ ***Resource Owner (User)**: You want to give PrintMyPhotos access to your photos without giving them your Google account password*

→ ***Third-Party Application**: This is the application that wants to access your photos to print them*

→ ***Resource Server**: This is the server that holds your photos and has the data that PrintMyPhotos wants to access.*

→ ***Authorization Server**: This server handles the authentication (logging in) and authorization (granting permissions) for Google services.*

→ ***Client**: This is the application that requests access to the resource server (Google Photos) on behalf of the user.*



1. You go to PrintMyPhotos and click on "Import Photos from Google Photos."

2.PrintMyPhotos sends you to Google's authorization server to log in and grant permission.

3. Google's authorization server asks you to log in (if you are not already logged in) and then asks if you want to give PrintMyPhotos permission to access your Google Photos.

4. You agree and grant permission. Google's authorization server then sends an authorization code back to PrintMyPhotos.

5. PrintMyPhotos sends the authorization code to Google's authorization server and requests an access token.

6. PrintMyPhotos uses this access token to request your photos from the Google Photos resource server.

Application Flow

Faisal Memon (EmbarkX)

Step	Description	Component	Action	Details
1	User Initiates OAuth2 Login	Login.js (React)	User clicks "Login with GitHub" or "Login with Google" button.	Redirects to: http://localhost:8080/oauth2/authorization/{provider}
2	Spring Security Handles the OAuth2 Redirect	application.properties (Spring Boot)	Defines OAuth2 client details (client ID, secret, redirect URI).	Sets up: OAuth2 client registration.
3	User is Redirected to OAuth2 Provider	Spring Security Endpoint: /oauth2/authorization/{registrationId}	Redirects user to OAuth2 provider's authorization page.	Provider: GitHub or Google.
4	User Authenticates with OAuth2 Provider	OAuth2 Provider's Authorization Page (External)	User logs in and authorizes the application.	Redirects back to: Application with authorization code.
5	Spring Security Exchanges Code for Access Token	Spring Security Built-in Logic	Exchanges authorization code for access token.	Retrieves: User profile information.

Step	Description	Component	Action	Details
6	OAuth2LoginSuccessHandler is Invoked	OAuth2LoginSuccessHandler.java (Spring Boot)	Handles successful authentication.	Tasks: 1. Check if user exists in the database. 2. Register new user if not exists. 3. Generate JWT token. 4. Redirect to frontend with JWT token.
7	User is Redirected to Frontend with JWT Token	Redirection to Frontend	Redirects user to a specific route in React (e.g., /oauth2/redirect).	Includes: JWT token in URL query parameters.
8	React Handles OAuth2 Redirect	OAuth2RedirectHandler.js (React)	Handles redirect and extracts JWT token.	Tasks: 1. Extract JWT token from URL. 2. Decode token to extract user information. 3. Store token and user info in local storage. 4. Update context state. 5. Redirect to protected route.
9	User Navigates to Protected Route	PrivateRoute.js (React)	Ensures only authenticated users can access protected routes.	Checks: Token in local storage.
10	Setting Up Routes in React	App.js (React)	Set up routes for Login, OAuth2RedirectHandler, and protected routes using PrivateRoute.	Routes: /login: Login page. /oauth2/redirect: Handles OAuth2 redirect. /home and /: Protected routes.

Importance of Custom Success Handler

Faisal Memon (EmbarkX)

Importance

→ *User Registration and Management*

→ *Security Context Update*

→ *JWT Token Generation*

→ *Custom Redirection*

What Happens if Not Defined

→ *No Custom User Handling*

→ *No JWT Token Generation*

→ *Default Redirection*

→ *Security Context*

What is Multi Factor Authentication

Faisal Memon (EmbarkX)

Multi-Factor Authentication (MFA) is a security process that requires you to prove your identity in multiple ways before accessing an account or system

How does it work?

***Something you know**: This is usually a password or a PIN.*

***Something you have**: This could be a smartphone, a security token, or a key card.*

***Something you are**: This involves biometric verification, like a fingerprint or facial recognition.*

Examples

→ *Banking Apps*

→ *Email Services*

→ *Social Media Platforms*

Thank you

Multi Factor Authentication Flow

Faisal Memon (EmbarkX)

Flow

Step	Action	Description
1. Login Attempt	Enter Username and Password	You start by entering your username and password on the login page of the service you want to access.
2. Initial Authentication	Verify Password	The service verifies your password (something you know). If correct, it proceeds to the next step.
3. Second Factor Request	Prompt for Second Factor	The service prompts you for a second factor, such as a code, push notification, or biometric scan.
4. Receiving the Second Factor	Receive Code or Notification, Perform Biometric Scan	Depending on the method, you receive a code via SMS/email/app or get a push notification or perform a scan.
5. Entering/Approving the Second Factor	Enter Code or Approve Notification, Complete Scan	You enter the received code, approve the push notification, or complete the biometric scan.
6. Verification of Second Factor	Service Verifies Second Factor	The service verifies the second factor. If it matches or is approved, it confirms your identity.
7. Access Granted	Gain Access to Account	Once both factors are verified, the service grants you access to your account.

Example

Step	**Action**	**Description**
1. Open the Banking App	Open the Banking App on your phone	You start by opening your banking app on your phone.
2. Enter Username and Password	Enter Username and Password	You enter your username and password.
3. Receive SMS Code	App Sends Code to Registered Phone	The app sends a code to your registered phone number.
4. Enter SMS Code	Enter Code from SMS	You enter the code from the SMS into the app.
5. Access Your Account	App Verifies Code and Grants Access	The app verifies the code and grants you access to your account.

General Flow of 2FA with Google Authenticator

Step	Action	Description
1. User Registration	User Registers an Account	The user signs up for an account by providing their details (e.g., email, password).
2. 2FA Setup Initiation	System Generates QR Code for Google Authenticator	After registration, the system generates a unique QR code linked to the user's account.
3. Scan QR Code	User Scans QR Code with Google Authenticator	The user scans the QR code using the Google Authenticator app on their smartphone.
4. Generate 2FA Code	Google Authenticator Generates a 6-digit Code	Google Authenticator generates a 6-digit code that changes every 30 seconds.
5. Enter 2FA Code	User Enters the Code from Google Authenticator	The user enters the 6-digit code from the Google Authenticator app into the Spring Boot application.
6. Verify 2FA Code	System Verifies the Entered Code	The Spring Boot application verifies the entered code against the one generated by Google Authenticator.

General Flow of 2FA with Google Authenticator

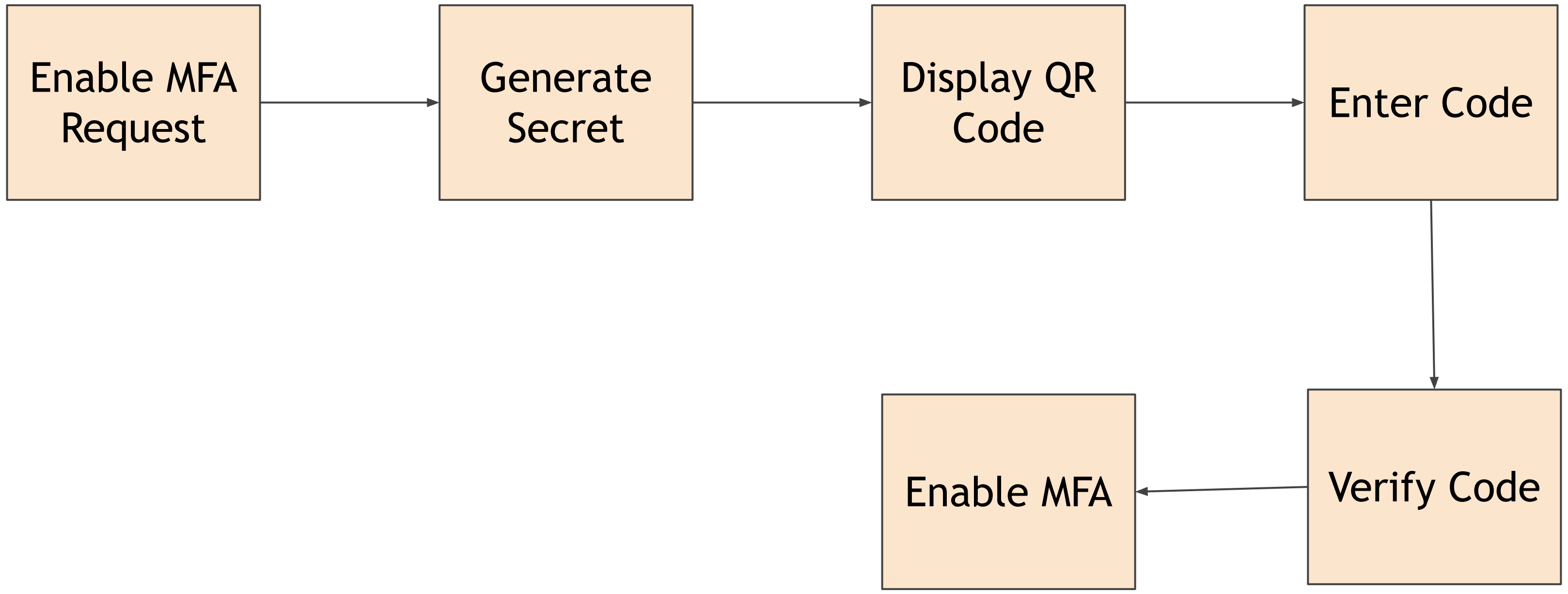
Step	Action	Description
7. Complete Registration	User Registration Completes if Code is Verified	If the code is verified successfully, the user's account is fully registered with 2FA enabled.
8. Login Attempt	User Enters Username and Password	The user attempts to log in by entering their username and password.
9. Prompt for 2FA Code	System Prompts User to Enter 2FA Code	After password verification, the system prompts the user to enter the 6-digit code from Google Authenticator.
10. Enter 2FA Code	User Enters the Code from Google Authenticator	The user enters the 6-digit code from the Google Authenticator app.
11. Verify 2FA Code	System Verifies the Entered Code	The system verifies the entered code against the one generated by Google Authenticator.
12. Access Granted	User Gains Access to the Account if Code is Verified	If the code is verified successfully, the user gains access to their account.

Steps for our application

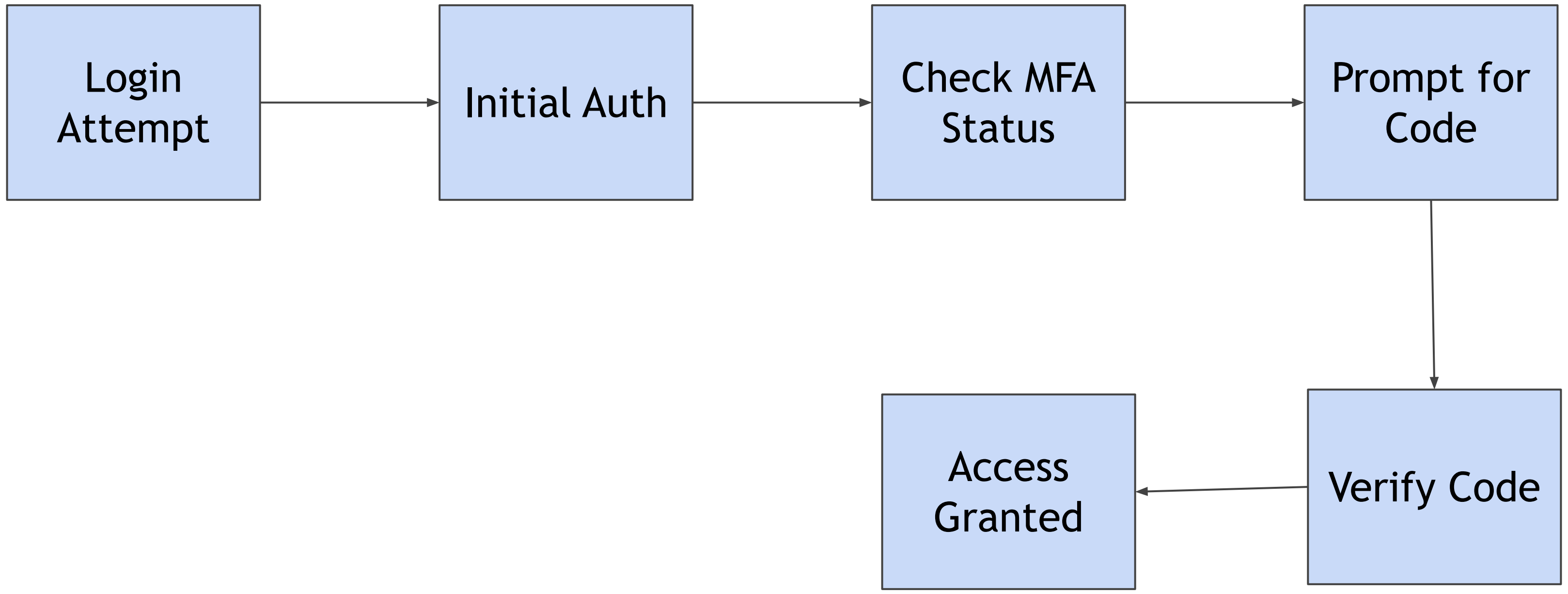
→ Enable MFA Flow

→ Login with MFA Flow

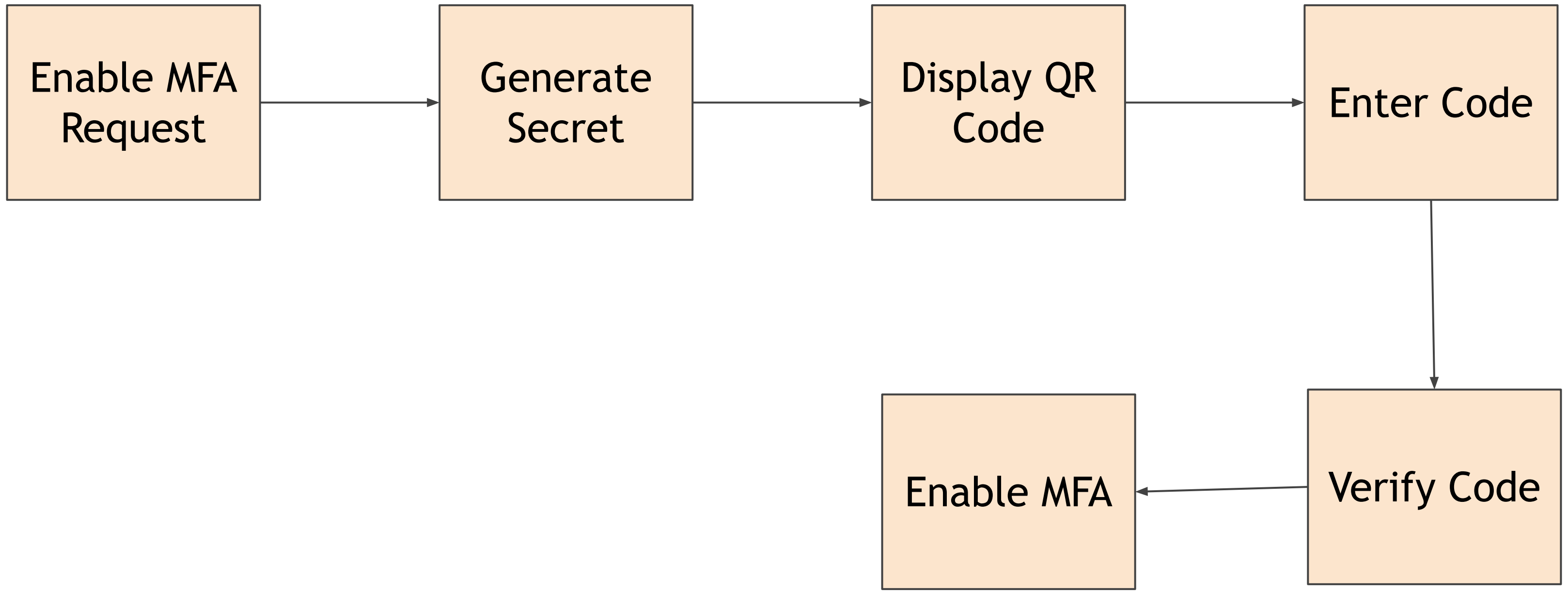
Enable MFA Flow



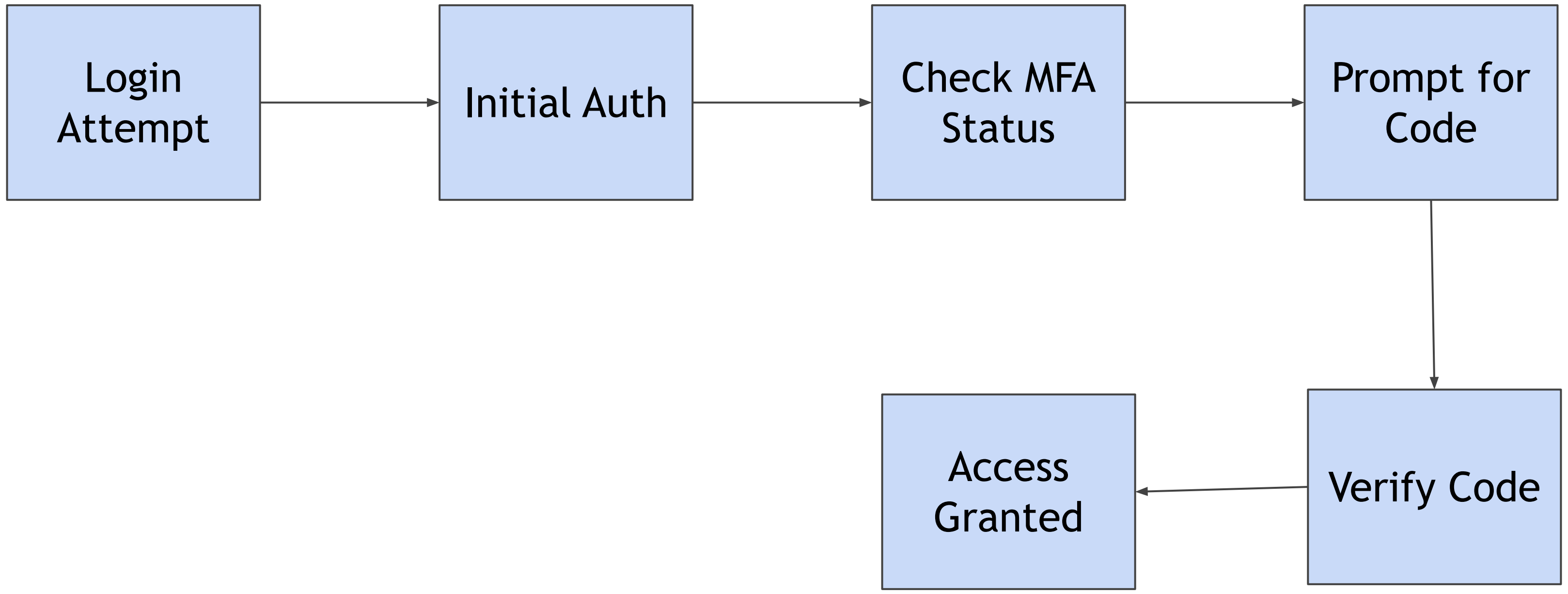
Login with MFA Flow



Enable MFA Flow



Login with MFA Flow



Setting up the Backend API's

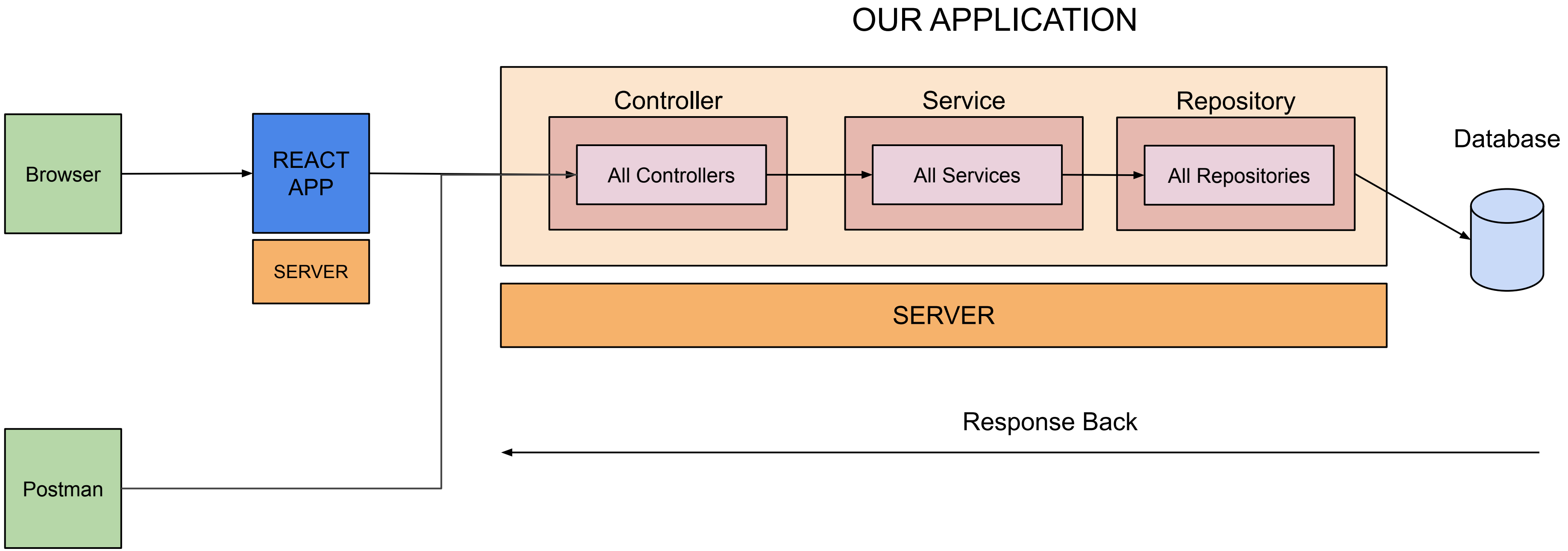
Faisal Memon (EmbarkX)

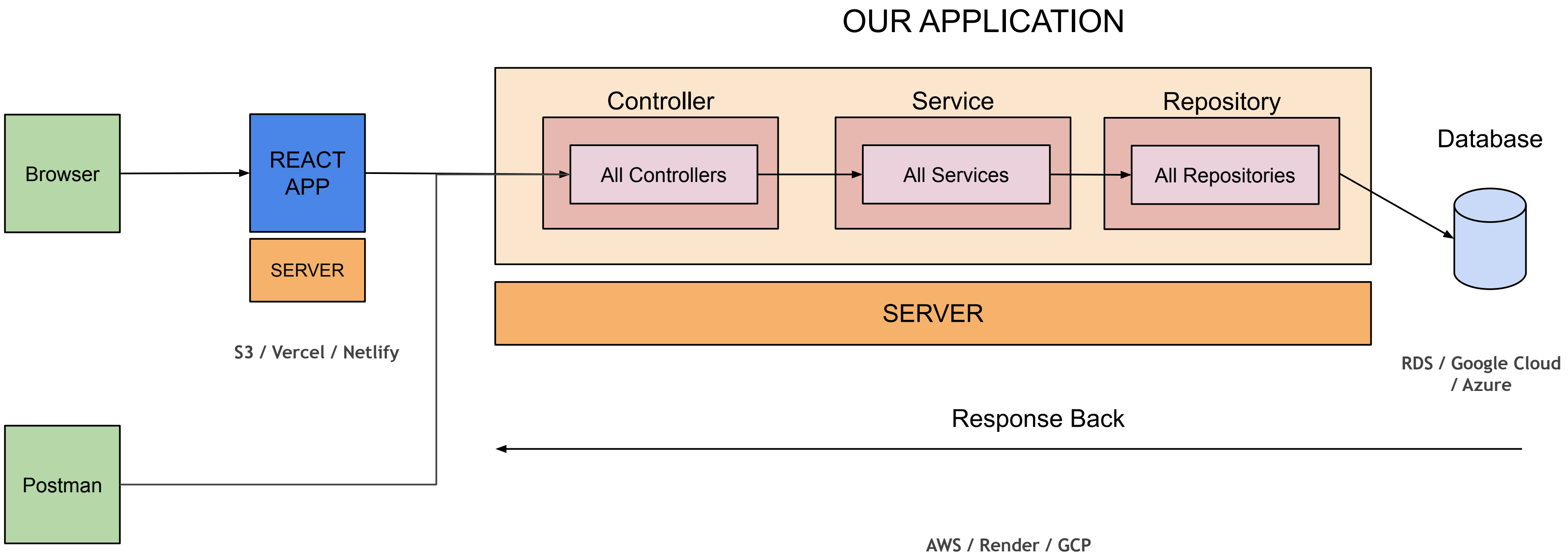
HTTP Method	Endpoint	Description	Request Parameters	Response
POST	/enable-2fa	Enables 2FA for the logged-in user	None	QR code URL for configuring 2FA
POST	/disable-2fa	Disables 2FA for the logged-in user	None	"2FA disabled"
POST	/verify-2fa	Verifies the 2FA code for the logged-in user	int code	"2FA verified" if valid, "Invalid 2FA code" if invalid
GET	/user/2fa-status	Gets the 2FA status for the logged-in user	None	JSON object with the 2FA status, or "User not found"
POST	/public/verify-2fa-login	Verifies the 2FA code during the login process using JWT	int code, String jwtToken	"2FA verified" if valid, "Invalid 2FA code" if invalid

Understanding Deployments and How It Works

Faisal Memon (EmbarkX)

Deployment is the process of moving software from the **development** environment to the **production** environment where it can be used by end-users.


OUR APPLICATION
Browser
REACT APP
SERVER
Postman
Controller
All Controllers
Service
All Services
Repository
All Repositories
Database
SERVER
Response Back


OUR APPLICATION
Browser
REACT APP
SERVER
S3 / Vercel / Netlify
Postman
Controller
All Controllers
Service
All Services
Repository
All Repositories
Database
SERVER
RDS / Google Cloud / Azure
Response Back
AWS / Render / GCP

If you think this course helped you, please do help provide an **honest rating and review** of the course. Your insights help us **improve and provide better content** for future learners.

We appreciate your **support** and look forward to hearing your **thoughts**!