Introduction to REST API Principles

KOENIG
step forward

Understanding REST Architecture

Understanding REST Architecture

- Principles of REST: Statelessness, client-server separation, uniform interfaces.
- HTTP methods: GET, POST, PUT, DELETE, PATCH, and their appropriate use cases.
- API design best practices: Resource naming, versioning, and HATEOAS.

What is REST Architecture?

- REST (Representational State Transfer) is an architectural style for designing networked applications.
- It is based on a set of principles that enable communication between clients (such as web browsers or mobile apps) and servers over the internet.
- RESTful systems use standard HTTP methods to perform operations on resources, making them simple, scalable, and stateless.
- The three key principles of REST are Statelessness, Client-Server Separation, and Uniform Interfaces.

Principles of REST: Statelessness, client-server separation, uniform interfaces.

- Statelessness
  - Each request from a client to a server must contain all the necessary information to process it.
  - The server does not store any client-specific session data between requests.
- Key Characteristics of Statelessness:
  - No session storage on the server.
  - Each request is independent and self-contained.
  - Clients must include authentication details (like API keys, JWT tokens) with every request.
  - Improves scalability since the server does not maintain session state.

Principles of REST: Statelessness, client-server separation, uniform interfaces.

- Client-Server Separation
  - The client (frontend) and server (backend) operate independently.
  - The client sends requests, the server processes them and responds with data.
- Key Characteristics of Client-Server Separation:
  - Independence: Clients and servers can evolve separately.
  - Scalability: Multiple clients (web, mobile) can consume the same REST API.
  - Security: Servers enforce authentication and authorization, keeping client-side logic separate.
- Example Architecture:
  - Client (React, Angular, iOS, Android) → Requests Data
  - Server (Node.js, Spring Boot) → Processes Requests and Returns Data

Principles of REST: Statelessness, client-server separation, uniform interfaces.

- Uniform Interface
  - A REST API should have a consistent and standardized way of accessing resources, making it predictable and easy to use.
- Key Constraints of Uniform Interface:
  - Resource Identification: Every resource has a unique URI (e.g., /users/1).
  - Standard HTTP Methods: RESTful APIs use:
    - GET → Retrieve data
    - POST → Create a new resource
    - PUT → Update an existing resource
    - DELETE → Remove a resource
  - Self-descriptive Messages: Responses should contain sufficient information for the client to process them.

HTTP methods: GET, POST, PUT, DELETE, PATCH, and their appropriate use cases.

- HTTP methods define the actions that can be performed on resources in a RESTful API.
- The most commonly used methods are GET, POST, PUT, DELETE, and PATCH.
- GET – Retrieve Data
  - Fetch data from the server without modifying it.
  - Use Cases:
    - Retrieve a list of resources (GET /products)
    - Retrieve a single resource (GET /products/1)
    - Fetch data without causing side effects

HTTP methods: GET, POST, PUT, DELETE, PATCH, and their appropriate use cases.

- PATCH – Update a Resource (Partial Update)
  - Modify specific fields of an existing resource instead of replacing the entire resource.
  - Use Cases:
    - Update only the price of a product (PATCH /products/1)
    - Modify user details without affecting other fields (PATCH /users/10)
- DELETE – Remove a Resource
  - Delete an existing resource from the server.
  - Use Cases:
    - Remove a user account (DELETE /users/10)
    - Delete a product from the inventory (DELETE /products/1)

API design best practices: Resource naming, versioning, and HATEOAS.

- A well-designed REST API is easy to use, scalable, and maintainable.
- The three key best practices for API design are resource naming, versioning, and HATEOAS (Hypermedia as the Engine of Application State).

Resource Naming Best Practices

- Proper resource naming ensures clarity and consistency in API endpoints.
- Best Practices for Naming Resources:
  - Use **nouns**, not verbs (Resources represent entities, not actions).
  - Use **plural names** for collections and singular for specific resources.
  - Use **hyphens (-)** to separate words, not underscores (GET /user-profile, not GET /user_profile).
  - Use **lowercase** letters for URLs (GET /users, not GET /Users).

Resource Naming Best Practices

- Examples of Well-Designed Resource Names

Resource	Correct	Incorrect
Users Collection	GET /users	GET /getUsers
Single User	GET /users/10	GET /users?id=10
User's Orders	GET /users/10/orders	GET /users/orders?id=10
Search (with query parameters)	GET /products?category=electronics	GET /searchProducts/electronics

API Versioning Best Practices

- Versioning allows you to introduce new features or modify existing ones without breaking client applications.
- Best Practices for Versioning
  - Use **explicit versioning** in the URL (/v1/, /v2/), not implicit.
  - Use **major versions** only (e.g., v1, v2), not minor versions (v1.1).
  - Provide **backward compatibility** when updating APIs.
  - Deprecate old versions with a clear timeline.

API Versioning Best Practices

- Common API Versioning Methods

Versioning Method	Example	Pros	Cons
URL Versioning	GET /v1/products	Easy to implement, clear	Requires updating URLs when version changes
Header Versioning	GET /products + Accept: application/vnd.company.v1+json	Clean URLs, flexible	Clients must send correct headers
Query Parameter Versioning	GET /products?version=1	Easy for testing	Messy, less commonly used

- **Recommended: URL versioning** (/v1/) for clarity and ease of use.

HATEOAS (Hypermedia as the Engine of Application State)

- HATEOAS enhances REST APIs by including links in responses, guiding clients on available actions.
- Best Practices for HATEOAS
  - Include links to related resources in API responses.
  - Use self-descriptive links (self, update, delete).
  - Reduce the need for hardcoded URLs in client applications.

HATEOAS (Hypermedia as the Engine of Application State)

- Example of a HATEOAS-Enabled Response

```
{
  "id": 1,
  "name": "Laptop",
  "price": 1200,
  "links": {
    "self": "/products/1",
    "update": "/products/1/update",
    "delete": "/products/1/delete"
  }
}
```

- Benefits: Clients dynamically discover available actions instead of relying on documentation.

Building REST APIs with Spring Boot

- Setting up a Spring Boot project.
- Creating RESTful endpoints and mapping HTTP methods to Java methods.
- Handling request parameters, path variables, and request bodies.

Setting up a Spring Boot project.

1. Go to Spring Initializer: https://start.spring.io/
2. Select Project Settings:
   - Project: Maven
   - Language: Java
   - Spring Boot Version: Choose the latest stable version
   - Packaging: Jar
   - Java Version: 21
3. Define Project Metadata:
   - Group: com.example
   - Artifact: my-spring-boot-app
   - Name: my-spring-boot-app
   - Package Name: com.example.myspringbootapp

Setting up a Spring Boot project.

4. Select Dependencies (for a basic MVC setup with MySQL):
   - Spring Web (for REST APIs)
   - Spring Boot DevTools (for auto-reloading)
5. Generate & Download the Project:
   - Click "Generate", then extract the ZIP file.
6. Open the Project in IntelliJ IDEA
   - Open IntelliJ IDEA Community Edition
   - Click "Open" and select the extracted my-spring-boot-app folder.
   - Wait for Maven dependencies to download.

Creating RESTful endpoints and mapping HTTP methods to Java methods.

- Create a Spring Boot REST Controller
  - Spring Boot uses the @RestController annotation to define RESTful APIs.

HTTP Method	Java Annotation	Example URL
GET	@GetMapping	/users
GET (by ID)	@GetMapping("/{id}")	/users/1
POST	@PostMapping	/users
PUT	@PutMapping("/{id}")	/users/1
PATCH	@PatchMapping("/{id}")	/users/1?name=Updated Name
DELETE	@DeleteMapping("/{id}")	/users/1

Handling request parameters, path variables, and request bodies.

- Handling Path Variables (@PathVariable)
  - Path variables are used to extract values from the URL path.
- Handling Request Parameters (@RequestParam)
  - Request parameters are used to pass optional or query parameters in the URL.
- Handling Request Body (@RequestBody)
  - Request bodies are used in POST, PUT, and PATCH requests to send JSON data.

Error Handling in REST APIs

- Custom exception handling in Spring Boot.
- Designing standard error responses.
- Implementing global exception handling using @ControllerAdvice.

Custom exception handling in Spring Boot.

- Spring Boot provides a structured way to handle exceptions in REST APIs using @ExceptionHandler, and @RestControllerAdvice
- Basic Exception Handling Using @ExceptionHandler
  - The @ExceptionHandler annotation is used inside a controller to handle specific exceptions.
- Global Exception Handling Using @RestControllerAdvice
  - Instead of handling exceptions per controller, you can use @RestControllerAdvice to centralize exception handling.

Designing standard error responses.

- A well-structured error response improves the API’s usability and helps clients debug issues effectively.
- Structure of a Standard Error Response
- A standard error response should include the following fields:
  - timestamp → When the error occurred
  - status → HTTP status code (e.g., 400, 404, 500)
  - error → HTTP status message (e.g., "Bad Request", "Not Found")
  - message → A human-readable error message
  - path → The requested endpoint that caused the error

Implementing global exception handling using @ControllerAdvice

- Global exception handling allows you to manage errors centrally, improving maintainability and user experience.
- Why Use @ControllerAdvice or @RestControllerAdvice?
  - Centralized error handling for all controllers.
  - Ensures consistent error response format.
  - Separates business logic from error handling.

Advanced REST API Development

Securing REST APIs

- Authentication and Authorization: OAuth2 and JWT basics.
- Implementing security in Spring Boot using Spring Security.
- Protecting APIs from common vulnerabilities (e.g., CSRF, XSS).

Security Concepts

- **Principal**
  - User, device or system that performs an action
- **Authentication**
  - Establishing that a principal's credentials are valid
- **Authorization**
  - Deciding if a principal is allowed to access a resource
- **Authority**
  - Permission or credential enabling access (such as a role)
- **Secured Resource**
  - Resource that is being secured

Authentication

- There are many authentication mechanisms
  - *Examples:* Basic, Digest, Form, X.509, OAuth 2.0 / OIDC
- There are many storage options for credential and authority data
  - *Examples:* in-memory (for development only), Database, LDAP

Authorization

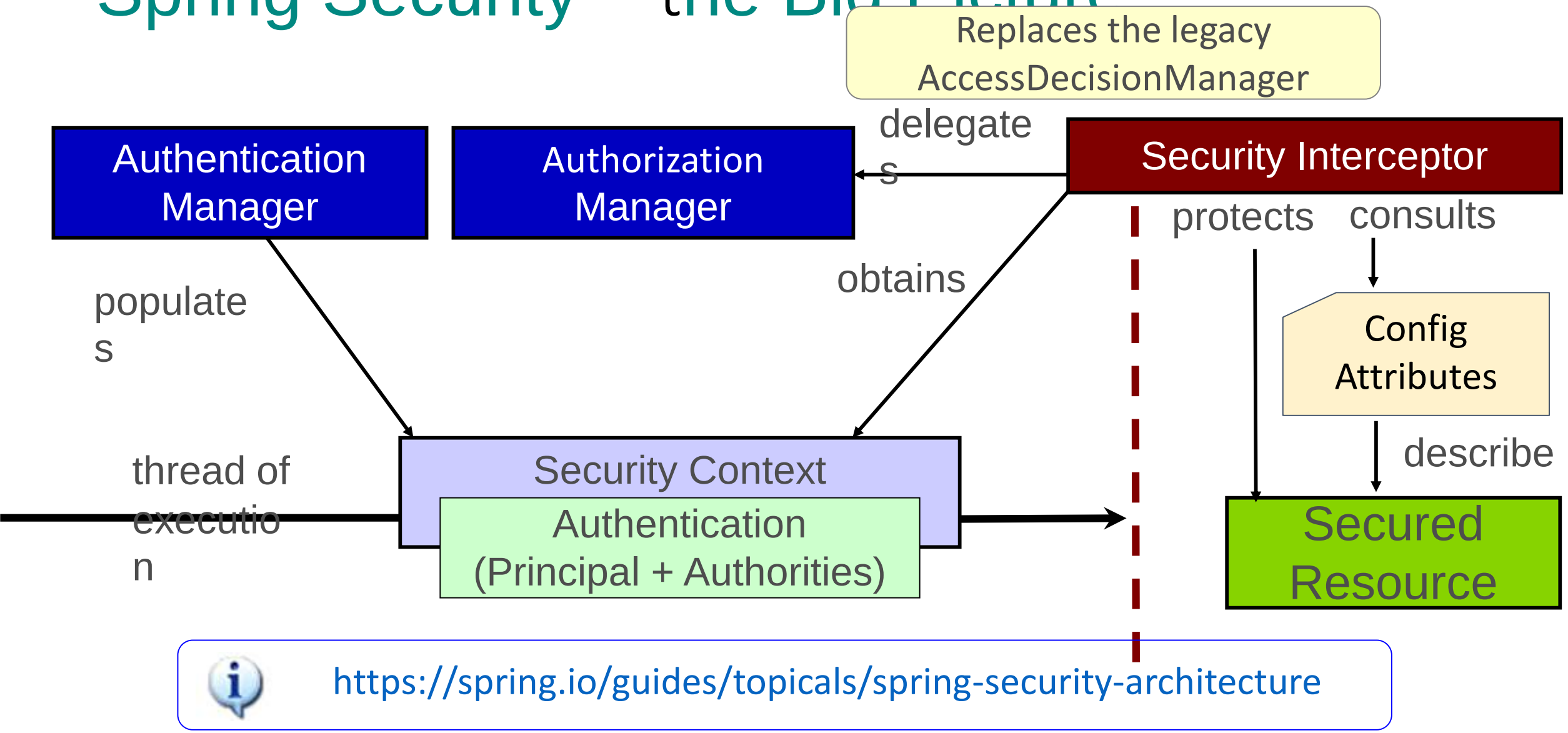
- Authorization depends on authentication
  - Before deciding if a user is permitted to access a resource, user identity must be established
- Authorization determines if you have the required *Authority*
- The decision process is often based on roles
  - *ADMIN* role can cancel orders
  - *MEMBER* role can place orders
  - *GUEST* role can browse the catalog

A *Role* is simply a commonly used type of *Authority.*

Spring Security

- Portable
  - Can be used on any Spring project
- Separation of Concerns
  - Business logic is *decoupled* from security concern
  - Authentication and Authorization are *decoupled*
    - Changes to authentication have *no impact* on authorization
- Flexible & Extensible
  - *Authentication:* Basic, Form, X.509, OAuth, Cookies, Single-Sign-On, …
  - *Storage:* LDAP, RDBMS, Properties file, custom DAOs, …
  - Highly customizable

Spring Security – the Big Picture



https://spring.io/guides/topicals/spring-security-architecture

Setup and Configuration
Spring Security in a Web Environment

Three steps

1. Setup Filter chain
2. Configure security (authorization) rules
3. Setup Web Authentication

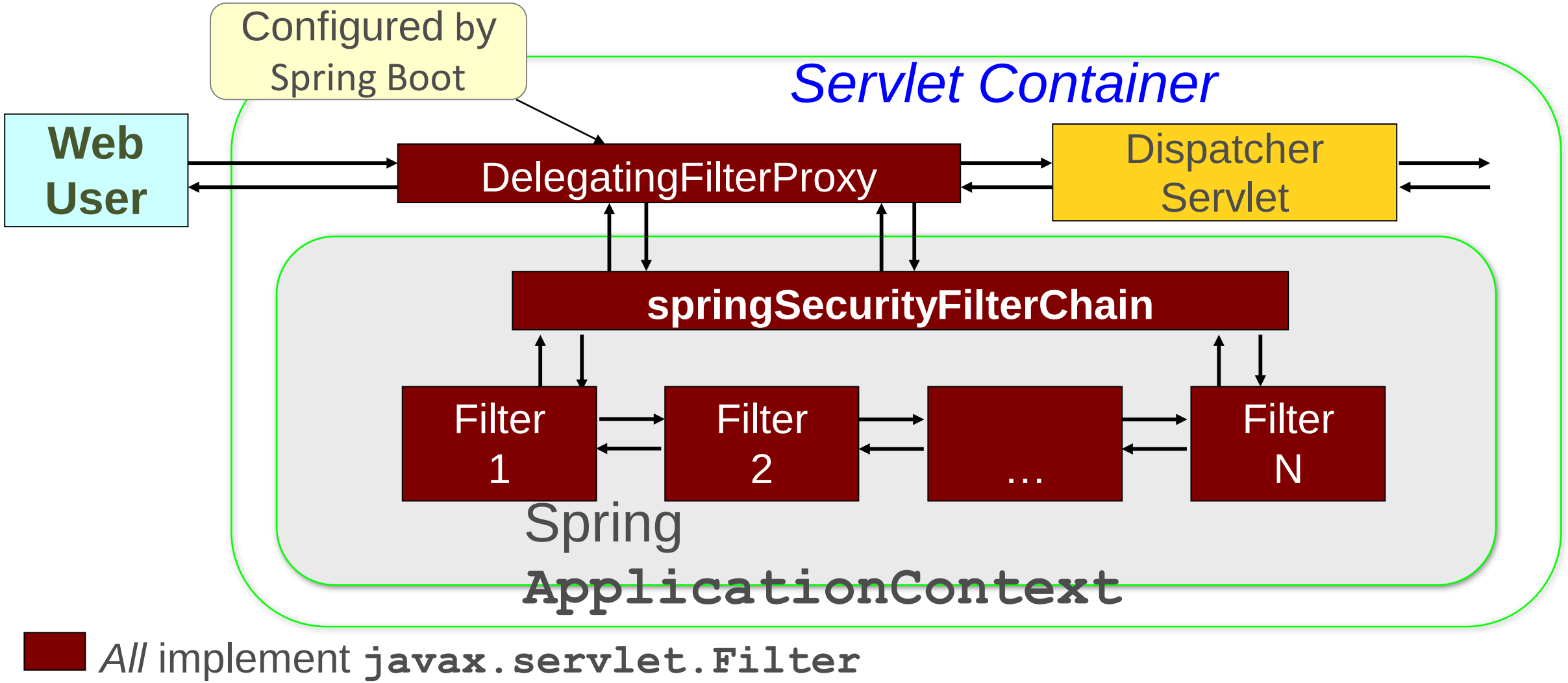
Spring Security is **not** limited to Web security, but that is all we will consider here, and it is configurable “out-of-the-box”

Spring Security Filter Chain – 1

- Implementation is a *chain* of Spring configured *filters*
  - Requires a **`DelegatingFilterProxy`**
    - Automatically configured by Spring Boot
  - Chain consists of many filters (next slide)

For more details see "*Advanced security: working with filters*" at end of this topic.

Spring Security Filter Chain – 2



All implement `javax.servlet.Filter`

Spring Security Filters

#	Filter Name	Main Purpose
1	`SecurityContextPersistenceFilter`	Establishes SecurityContext and maintains between HTTP requests
2	`LogoutFilter`	Clears SecurityContextHolder when logout requested
3	`UsernamePasswordAuthenticationFilter`	Puts Authentication into the SecurityContext on login request.
4	`ExceptionTranslationFilter`	Converts SpringSecurity exceptions into HTTP response or redirect
5	`AuthorizationFilter`	Authorizes web requests based on config attributes and authorities

Spring Boot Default Security Setup

- Sets up a single in-memory user called “user”
- Auto-generates a UUID password
- Relies on Spring Security’s content-negotiation strategy to determine whether to use httpBasic or formLogin
- All URLs require a logged-in user

```
INFO : o.s.b.web.servlet.FilterRegistrationBean - Mapping filter: 'httpTraceFilter' to: [/*]
INFO : o.s.b.web.servlet.FilterRegistrationBean - Mapping filter: 'webMvcMetricsFilter' to: [/*]
INFO : o.s.b.w.servlet.ServletRegistrationBean - Servlet dispatcherServlet mapped to [/]
INFO : o.s.b.a.w.s.WelcomePageHandlerMapping - Adding welcome page: class path resource [static/index.html]
INFO : o.s.b.a.s.s.UserDetailsServiceAutoConfiguration -

Using generated security password: f49a49f1-df8a-4da8-b3e8-89fb204bda24

INFO : o.s.s.web.DefaultSecurityFilterChain - Creating filter chain: org.springframework.security.web.util.matcher.AnyRequ
INFO : o.s.b.d.a.OptionalLiveReloadServer - LiveReload server is running on port 35729
```

Spring Security Configuration

`WebSecurityConfigurerAdapter` is deprecated as of Spring Security 5.7/Spring Boot 2.7

```
@Configuration
public class SecurityConfig {

   @Bean
   public SecurityFilterChain filterChain(HttpSecurity http) throws Exception {

   }

   @Bean
   public InMemoryUserDetailsService userDetailsService() {

   }
}
```

Configures the filter chain

Configures the AuthenticationManager

Authorizing URLs

- Define specific authorization restrictions for URLs
- Uses the Spring MVC matching rules if available, otherwise uses "*Ant-style*" pattern matching
  - **`"/admin/*"`** only matches **`"/admin/xxx"`**
  - **`"/admin/**"`** matches *any* path under **`/admin`**

```
@Bean
public SecurityFilterChain filterChain(HttpSecurity http) throws Exception {
   http.authorizeHttpRequests((authz) -> authz
         .requestMatchers("/admin/**").hasRole("ADMIN")
         …
}
```

Match *all* URLs starting with **/admin**

User must have **ADMIN** role

More on `authorizeRequests()`

- *Chain* multiple restrictions - evaluated in the order listed
  - First match is used, ***put specific matches first***

```
@Bean
public SecurityFilterChain filterChain(HttpSecurity http) throws Exception {
   http.authorizeHttpRequests((authz) -> authz
        .requestMatchers("/signup", "/about").permitAll()
           .requestMatchers(HttpMethod.PUT,"/accounts/edit*").hasRole("ADMIN")
           .requestMatchers("/accounts/**").hasAnyRole("USER","ADMIN")
           .anyRequest().authenticated());
   return http.build();
}
```

Must be authenticated for any other request

Spring Security supports *roles* out-of-the-box – but *there are **no** predefined roles.*

Warning: URL Matching

- Older code may use **`antMatchers`** / **`mvcMatchers`**

```
http.authorizeHttpRequests((authz) -> authz
    // Only matches /admin
    .antMatchers("/admin").hasRole("ADMIN")
    // Matches /admin, /admin/
    .mvcMatchers("/admin").hasRole("ADMIN"))
```

These matchers are deprecated in Spring Security 5.8

- Use **`requestMatchers`**
  - Uses the most appropriate `RequestMatcher`
  - Newer API, more secure defaults, *recommended*

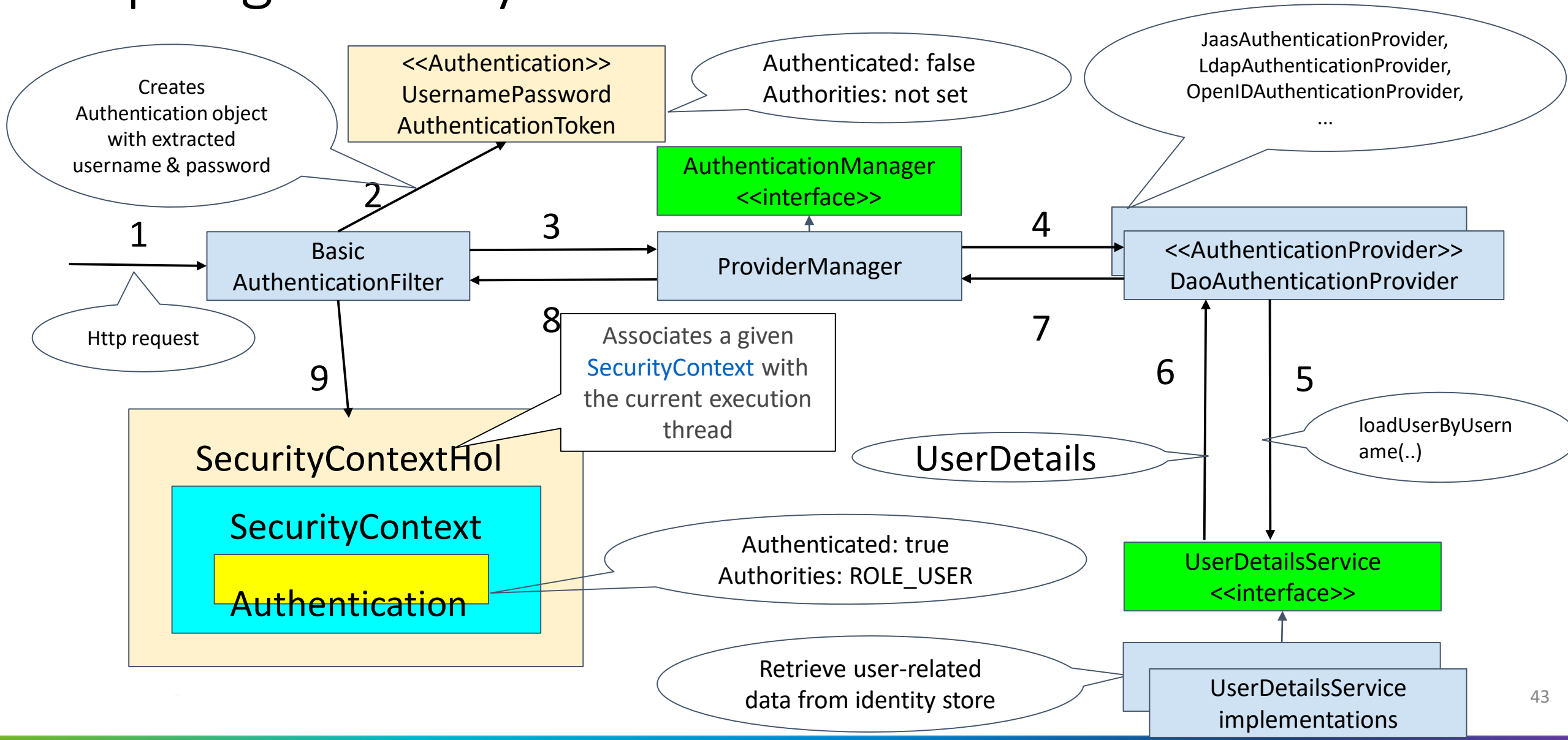
By-passing Security

- Some URLs need not be secured (such as static resources)
  - `permitAll()` allows open-access
    - But still processed by Spring Security Filter chain
- Can bypass Security completely

```
@Bean
public WebSecurityCustomizer webSecurityCustomizer() {
        return (web) -> web.ignoring().requestMatchers("/ignore1", "/ignore2");
}
```

These URLs pass straight through, no checks

Spring Security Authentication Flow



AuthenticationProvider & UserDetailsService

- Out-of-the-box **`AuthenticationProvider`** implementations
  - **`DaoAuthenticationProvider, LdapAuthenticatonProvider, OpenIDAuthenticationProvider, RememberMeAuthenticationProvider, etc.`**
- **`DaoAuthenticationProvider`** retrieves user details from a configured **`UserDetailsService`**
- Out-of-the-box **`UserDetailsService`** implementations
  - **`InMemoryUserDetailsManager`** uses in-memory identity store
  - **`JdbcUserDetailsManager`** uses database identity store
  - **`LdapUserDetailsManager`** uses Ldap identity store

In-Memory UserDetailsService

- Example of a built-in **`UserDetailsService`**
  - **`InMemoryUserDetailsManager`** implements **`UserDetailsService`** interface & **`UserDetailsManager`** interface

```
@Bean
public InMemoryUserDetailsManager userDetailsService() {

 UserDetails user =

User.withUsername("user").password(passwordEncoder.encode("user")).roles("USER").
build();

 UserDetails admin =
```

login

password

Supported roles

Database UserDetailsService – 1

- Another example of a built-in **`UserDetailsService`**
  - **`JdbcUserDetailsManager`** extends **`JdbcDaoImpl`** which implements the **`UserDetailsManager`** interface

```
@Bean
public UserDetailsManager userDetailsManager(DataSource dataSource) {
    return new JdbcUserDetailsManager(dataSource);
}
```

Sets up JdbcUserDetailsManager as UserDetailsService

Database UserDetailsService – 2

Queries RDBMS for users and their authorities

- Provides default queries
  - SELECT username, password, enabled FROM users WHERE username = ?
  - SELECT username, authority FROM authorities WHERE username = ?
- Groups also supported
  - **`groups`**, **`group_members`**, **`group_authorities`** tables
  - See online documentation for details

Implementing custom authentication

- Option #1: Implement custom **`UserDetailsService`** (using pre-configured **`DaoAuthenticationProvider`**)

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

- } Option #2: Implement custom **`AuthenticationProvider`**

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

Password Encoding

- Password must be stored in an encoded form
  - You cannot store password in plaintext form
- One-way transformation
  - You cannot decode it back to plaintext form
  - Authentication process compares user-provided password against the encoded one in the storage
- Spring Security supports multiple encoding schemes
  - *MD5PasswordEncoder* (Deprecated)
  - *SHAPasswordEncoder* (Deprecated)
  - *BCryptPasswordEncoder* (Currently recommended)

DelegatingPasswordEncoder to the Rescue

- Uses new password storage format: *{id}encodedPassword*
  - {id} = PasswordEncoder used to encrypt password
- Delegates to another PasswordEncoder based upon {id}
- BCrypt is current default

```
@Bean
public InMemoryUserDetailsManager userDetailsService() {
  PasswordEncoder encoder  =
PasswordEncoderFactories.createDelegatingPasswordEncoder();

  UserDetails user =

User.withUsername("user").password(passwordEncoder.encode("user")).roles("USER")
.build();
```

```
Generates {bcrypt}$2a$10$qfHYt54ZGLkHH4/SXgvPiudiNR5s.5bXX0QtTSTvLNyK8/aGec4s2
```

Enabling HTTP Authentication - 1

- Use the **`HttpSecurity`** object again
  - *Example:* HTTP Basic

```
@Bean
public SecurityFilterChain filterChain(HttpSecurity http) throws Exception {
   http.authorizeHttpRequests((authz) -> authz
           .requestMatchers("/admin/**").hasRole("ADMIN")
               .requestMatchers("/accounts/**").hasAnyRole("USER","ADMIN")
               .anyRequest().authenticated())
       .httpBasic(withDefaults()); //  Enable HTTP Basic

   return http.build();
}
```

Browser will prompt for username & password

Enabling HTTP Authentication - 2

Form based login

```
@Bean
public SecurityFilterChain filterChain(HttpSecurity http) throws Exception {
   http.authorizeHttpRequests((authz) -> authz
            .requestMatchers("/accounts/**").hasRole("USER")
            ...
        )
        .formLogin(form -> form       //  setup form-based authentication
                    .loginPage("/login") //  URL to use when login is needed
                    .permitAll()          //  any user can access
        )
        .logout(logout -> logout                  //  configure logout
                    .logoutSuccessUrl("/home")  //  go here after successful logout
                    .permitAll()                  //  any user can access
        );
   return http.build();
}
```

Default: **`/login?logout`**

An Example Login Page

URL that indicates an authentication request.
Default: POST to same URL used to display the form.

```
<form action="/login" method="POST">
  <input type="text" name="username"/>
  <br/>
  <input type="password" name="password"/>
  <br/>
  <input type="submit" name="submit" value="LOGIN"/>
</form>
```

login.html

The expected keys for generation of an authentication request token

Protecting APIs from common vulnerabilities (e.g., CSRF, XSS).

- What is CSRF?
  - CSRF is an attack where a malicious website tricks users into making unwanted requests to your API while authenticated.
- Solution: Enable CSRF Protection
  - Spring Security enables CSRF protection by default.
  - However, for stateless APIs (JWT-based authentication), we usually disable CSRF since tokens protect against CSRF.
  - Default CSRF protection (for form-based authentication):

    ```
    http.csrf(csrf -> csrf.enable()) // Keep CSRF enabled for session-based authentication
    ```
  - Disable CSRF for JWT-based authentication:

    ```
    http.csrf(csrf -> csrf.disable()) // Disable for JWT authentication
    ```

Protecting APIs from common vulnerabilities (e.g., CSRF, XSS).

- Preventing XSS (Cross-Site Scripting)
- What is XSS?
  - XSS allows attackers to inject malicious JavaScript into web applications, which can steal sensitive information.
- Solution: Use Content Security Policy (CSP)
  - Spring Boot can set CSP headers to prevent XSS.
  - Modify your security configuration:

```
http.headers(headers -> headers
        .contentSecurityPolicy(csp -> csp.policyDirectives("default-src 'self'"))
    )
```

Protecting APIs from common vulnerabilities (e.g., CSRF, XSS).

- Preventing SQL Injection
- What is SQL Injection?
  - SQL injection occurs when user inputs are directly concatenated into SQL queries, allowing attackers to execute harmful queries.
- Solution: Use Prepared Statements
  - Instead of:

    ```
    @Query("SELECT * FROM users WHERE username = '" + username + "'")
    ```

  - Use:

    ```
    @Query("SELECT u FROM User u WHERE u.username = :username")
    User findByUsername(@Param("username") String username);
    ```

  - Spring Data JPA automatically prevents SQL injection when using @Query with parameters.

Data Serialization and Validation

- Using Jackson for JSON serialization and deserialization.
- Validating API requests with @Valid and custom annotations.

Using Jackson for JSON serialization and deserialization.

- Jackson is the default JSON processing library used by Spring Boot for serializing Java objects to JSON and deserializing JSON to Java objects.
- Adding Jackson to a Spring Boot Project
  - Spring Boot automatically includes Jackson when using spring-boot-starter-web.
- JSON Serialization (Java Object → JSON)
  - When we return Java Object as response, Spring Boot automatically converts it to JSON.
- JSON Deserialization (JSON → Java Object)
  - When we send JSON data in a request, Spring Boot automatically converts it into a Java object.

Validating API requests with @Valid and custom annotations.

- Using @Valid for Request Validation
- Spring Boot integrates Jakarta Bean Validation (formerly Javax Validation) to validate API request payloads.
- Add Validation Dependency

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-validation</artifactId>
</dependency>
```

- Validating Request Data with @Valid
  - Spring Boot uses @Valid to apply validation constraints defined in the model.

Validating API requests with @Valid and custom annotations.

- Creating Custom Validation Annotations
  - Spring Boot allows custom validation annotations when built-in ones are not enough.
- Example: Custom Annotation to Validate Product Name
  - Create the Annotation (@ValidProductName)

```
@Documented
@Constraint(validatedBy = ProductNameValidator.class)
@Target({ElementType.FIELD})
@Retention(RetentionPolicy.RUNTIME)
public @interface ValidProductName {
   String message() default "Product name must start with an uppercase letter";
   Class<?>[] groups() default {};
   Class<? extends Payload>[] payload() default {};
}
```

Validating API requests with @Valid and custom annotations.

- Create the Validator Class

```
public class ProductNameValidator implements ConstraintValidator<ValidProductName, String>
{
   @Override
   public boolean isValid(String name, ConstraintValidatorContext context) {
      return name != null && Character.isUpperCase(name.charAt(0));
   }
}
```

- Global Exception Handling for Validation Errors
  - Spring Boot automatically throws MethodArgumentNotValidException when @Valid fails.
  - To return a custom error response, use @RestControllerAdvice.

Optimizing REST APIs

- Pagination and filtering for large datasets.
- Caching responses to improve performance.
- Using asynchronous processing for long-running requests.

Pagination and filtering for large datasets.

- When handling large datasets in REST APIs, fetching all records at once can lead to performance issues and high memory usage.
- Pagination and filtering help optimize API responses, improving efficiency and user experience.
- Implementing Pagination & filtering in Spring Boot
  - Spring Boot supports pagination using Spring Data JPA's Pageable interface.
  - Add Spring Data JPA Dependency (If Missing)

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-data-jpa</artifactId>
</dependency>
```

Caching responses to improve performance.

- Caching helps reduce database queries and response times by storing frequently accessed data in memory.
- Spring Boot provides caching support via Spring Cache Abstraction, with implementations like EhCache, Redis, Caffeine, and more.
- Enabling Caching in Spring Boot
  - Spring Boot requires enabling caching at the application level.
    - Add @EnableCaching in the Main Class
  - Caching API Responses using @Cacheable
    - @Cacheable stores method responses in cache so repeated calls return cached data instead of querying the database.
    - Apply Caching to the Service Layer

Caching responses to improve performance.

- Clearing Cache with @CacheEvict
  - When a product is added, updated, or deleted, we need to invalidate cache so new data is retrieved.
  - Cache is invalidated when a new record is added. The next request fetches fresh data from the database.
- Summary of Caching Annotations

Annotation	Description
@EnableCaching	Enables caching in the Spring Boot app
@Cacheable(value = "cacheName")	Caches method results
@CacheEvict(value = "cacheName", allEntries = true)	Clears cache on data update

Using asynchronous processing for long-running requests.

- Long-running requests can slow down API response times, leading to poor user experience and server overload.
- Asynchronous processing allows Spring Boot APIs to handle requests in the background while freeing up resources for other tasks.
- The @Async annotation allows methods to execute asynchronously in a separate thread.
- The method executes asynchronously without blocking the main thread.
- CompletableFuture<String> returns a future result when processing completes.

Testing and Documentation

- Writing unit tests for REST APIs using JUnit and Mockito.
- Automating API testing with Postman and REST Assured.
- Generating API documentation with Swagger/OpenAPI.

Writing unit tests for REST APIs using JUnit and Mockito.

- Unit testing ensures that individual components work as expected.
- For testing Spring Boot REST APIs, we use JUnit and Mockito to mock dependencies.
- Dependencies for Testing

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-test</artifactId>
    <scope>test</scope>
</dependency>
```

- spring-boot-starter-test includes JUnit and Mockito.

Automating API testing with Postman and REST Assured.

- Automating API testing ensures the stability of your REST endpoints.
- Automating API Testing with Postman
  - Step 1: Install Postman
    - Download and install Postman.(https://www.postman.com/downloads/)
  - Step 2: Create a Collection
    - Open Postman → Click "New Collection" → Name it "Product API Tests".
    - Add requests like GET /products, POST /products, etc.
  - Step 3: Add a Test Script
    - Open the request (GET /products).
    - Click "Tests" and add this script
  - Step 4: Run Automated Tests
    - Click Runner → Select "Product API Tests" → Run.
  - Postman Collection Runner executes multiple API tests at once.

```
pm.test("Status code is 200", function () {
    pm.response.to.have.status(200);
});
```

Automating API testing with Postman and REST Assured.

- Automating API Testing with REST Assured (Java)
  - REST Assured is a Java library for testing REST APIs.
  - Add REST Assured Dependency

```
<dependency>
    <groupId>io.rest-assured</groupId>
    <artifactId>rest-assured</artifactId>
    <scope>test</scope>
</dependency>
```

Generating API documentation with Swagger/OpenAPI.

- Swagger (OpenAPI) is a powerful tool for documenting REST APIs, making them easy to understand, test, and consume.
- Add Swagger Dependencies

```
<dependency>
  <groupId>org.springdoc</groupId>
  <artifactId>springdoc-openapi-starter-webmvc-ui</artifactId>
  <version>2.8.5</version>
</dependency>
```

- Enable OpenAPI in Spring Boot
  - Spring Boot automatically configures Swagger when the dependency is added.
  - You can access the documentation at: http://localhost:8080/swagger-ui/index.html