Secure Coding Practices and Guidelines

1. Overview

Secure coding practices and guidelines is a top level priority for any enterprise-level Java application. Every application should be developed with a strong focus on security, minimizing vulnerabilities and protecting sensitive data and systems. Regular updates, continuous security testing, and adherence to industry standards are essential components of a robust security strategy. Further it may involves a combination of techniques, tools, and processes to prevent vulnerabilities and attacks.

The purpose of this document is to list down the recommended Coding Practices and Guidelines mainly focused on application security. Formation of these practices is vital in the software development life cycle.

---

2. Input Validation

Input validation is a fundamental security measure that prevents malicious input from compromising the application.

2.1. Data Validation - Current Approach

- Validate all client provided data before processing, including all parameters in request body, URLs and HTTP header content such as names of cookies and values.
- Current systems are handling data validations in Java using Spring Boot Starter Validation library. Refer Java Coding Standards | 7. Input Validations

2.2. Whitelisting

- Currently, the inputs are validated for whitelisted characters, by regex expressions.

e.g.

Parameter	Regex Pattern	Whitelisted characters
URL	`^$\|^https?://(?:www\\.)?[\\w\\d\\-]+(?:\\.[\\w\\d\\-]+)+[\\w\\d\\-.,@?^=%&:/~+#]*[\\w\\d\\-@?^=%&/~+#]$`	. , @ ? ^ = % & : \ / ~ + # -

- Validate all String input against a list of allowed whitelisted characters, whenever possible. The characters may be different based on the parameter.
- If any potentially hazardous characters must be allowed as input, use proper encoding mechanisms (Refer 3.Input/Output Encoding) and modify the relevant regex patterns of the parameters.
- Examples of common hazardous characters include: ? & # = % + < > ' “ ( ) \ /

2.3. Regex Patterns

Parameter	Pattern
Address Line 1	`^([-''()_,.|`\"\\&](\\s)*)*[a-zA-ZÀ-ÖØ-öø-ÿ0-9-\\u1E00-\\u1EFF]+((\\s)*[a-zA-ZÀ-ÖØ-öø-ÿ0-9\\u1E00-\\u1EFF-''()_,./`|\"\\&]+)*$`

Address Line 2	`^$\|^([-''()_,.\|`\"\\&](\\s)*)*[a-zA-ZÀ-ÖØ-öø-ÿ0-9-\\u1E00-\\u1EFF]+((\\s)*[a-zA-ZÀ-ÖØ-öø-ÿ0-9\\u1E00-\\u1EFF-''()_,./`\|\"\\&]+)*$`
Area	`^[A-Z_]+$`
Country code	`^([A-Z]{2,3}(-[A-Z]{2})*,)*[A-Z]{2,3}(-[A-Z]{2})*$`
Country Name	`^[A-Z_ ]+$`
Cookie Value	`^[a-zA-Z0-9-]+$`
Discount code	`^$\|^[a-zA-Z0-9_]+$`
From Location	`^[A-Z0-9]+$`
Optional Extra Code	`^[A-Z_]+$`
Postcode	`^[a-zA-Z0-9_ ]+( [a-zA-Z0-9_ ]+)*$`
Telephone Number	`^(\\+?\\s*\\d\\s*){10,16}$`
Traveler Name	`([\\sA-Za-z\\u00C0-\\u024F\\u1E00-\\u1EFF-']+[A-Za-z\\u00C0-\\u024F\\u1E00-\\u1EFF-'\\s]*)$`
URL	`^$\|^https?://(?:www\\.)?[\\w\\d\\-]+(?:\\.[\\w\\d\\-]+)+[\\w\\d\\-.,@?^=%&:/~+#]*[\\w\\d\\-@?^=%&/~+#]$`

2.4. Data Validation - OpenApi Specification

Current Staysure Web(1.5) backend set up with BFF layer has the implementation of Swagger. Refer Swagger 3 (open api 3) and Swagger

Going forward, when we introduce 'API First Approach' with Microservices architecture, OpenApi 3.1 would be utilized to validate all client provided data, under the criterion such as required attributes, data type, minimum, maximum, maximum length of Strings, patterns, data formats and enums. Model classes could be automatically generated with the specified

validation rules. Existing implementations of Spring Bean validations could be minimized by this approach. However, custom validations can still be introduced and used with Spring Boot Validation and OpenApi Specification.

Examples:

- Verify all required attributes are present

```yaml
components:
  schemas:
    Organiser:
      type: object
      required:
        - firstName
        - lastName
        - dob
        - email
```

- Validate for expected data types

```yaml
        address:
          type: object
          properties:
            line1:
              type: string
              maxLength: 100
            line2:
              type: string
              maxLength: 100
            city:
              type: string
              maxLength: 50
```

- Validate data range specifying minimum and maximum values

```yaml
        expiryMonth:
          type: integer
          format: int32
          minimum: 1
          maximum: 12
```

- Validate data length for Strings

```yaml
        firstName:
          type: string
          maxLength: 100
```

- Validate data formats like dates

```yaml
      properties:
        startDate:
          type: string
          format: date
        endDate:
          type: string
          format: date-time
```

- Provide list of values for enums

```
        currency:
          type: string
          enum: [USD, EUR, GBP]

        policyPackage:
          type: string
          enum: [BASIC, COMPREHENSIVE, SIGNATURE, CLASSIC, DELUXE, ESSENTIALS]

```

- Validate Strings using Regex patterns where applicable

```
        firstName:
          type: string
          pattern: ([\\sA-Za-z\\u00C0-\\u024F\\u1E00-\\u1EFF-']+[A-Za-z\\u00C0-
\\u024F\\u1E00-\\u1EFF-'\\s]*)$
```

---

3. Input/Output Encoding

Output encoding is a critical security practice in Java applications and web development that involves transforming data into a safe format before rendering it in a user interface or output destination. This transformation ensures that any potentially harmful content is displayed in a way that doesn't compromise security or functionality.

3.1. JSON encoding

When micro services exchange data in the form of JSON payloads, encoding ensures that special characters are preserved and not misinterpreted as control characters or commands. **JSON** is the default encoding and **Jackson library** is the default JSON processor in Spring Boot, for automatic serialization and encoding of Java objects. This is used in all the applications by default since all are Spring Boot based applications.

```
@GetMapping(value = "/users", produces = MediaType.APPLICATION_JSON_VALUE)
public ResponseEntity<List<User>> getUsers() {
    // Implementation
}
```

3.2. XML encoding

Avoid XML encoding always, but if XML must be used, content negotiation must be provided.

```
@GetMapping(value = "/users", produces = MediaType.APPLICATION_XML_VALUE)
public ResponseEntity<List<User>> getUsersXml() {
    // Implementation
}
```

3.3. HTML encoding

Encoding HTML helps to ensure that any potentially harmful user input is rendered as text rather than executable code. Encoding converts special HTML characters into their corresponding HTML entities. For example, `<` becomes `<`, `>` becomes `>`, and `&` becomes `&`. This prevents HTML and JavaScript injection.

- If the API returns HTML content, ensure that special characters are encoded to prevent XSS attacks.
- When returning HTML content, set the `Content-Type` header to `text/html`.
- Use **OWASP Java Encoder** where user input such as comments, forums, or form inputs should be encoded to display safely in HTML.

```
import org.owasp.encoder.Encode;

public class HTMLEncoder {
    public void encodeInput(String userInput) {
        userInput = "<script>alert('XSS');</script>";
        String encoded = Encode.forHtml(userInput);
        System.out.println(encoded); // Outputs:
&lt;script&gt;alert(&#39;XSS&#39;);&lt;/script&gt;
    }
}
```

- Use **Jsoup Java HTML Parser** to sanitize user-generated HTML input such as comments, forums, or form inputs, by removing or escaping dangerous elements .

```
import org.jsoup.Jsoup;
import org.jsoup.safety.Safelist;

public class HtmlSanitizer {

    public static String sanitize(String unsafeHtml) {
        // Use Jsoup to clean the HTML input
        return Jsoup.clean(unsafeHtml, Safelist.basic());
    }

    public void sanitizeInput(String userInput) {
        userInput = "<script>alert('XSS Attack');</script><p>This is a paragraph.</p>";
        String safeOutput = sanitize(userInput);
        System.out.println(safeOutput); // Output: <p>This is a paragraph.</p>
    }
}

```

3.4. URL encoding

Ensures that URL components are transmitted correctly by converting special characters into a percent (%) encoded format.

- Space: %20
- Ampersand: %26
- Question Mark: %3F

Reserved characters like `/`, `:`, `&`, and `=` have special meanings in URLs and must be encoded if they appear in parameter values.

```
String queryParam = URLEncoder
.encode("https://example.com/api/books?title=Harry Potter & the Sorcerer's Stone&author=J.K. Rowling
", StandardCharsets.UTF_8);
```

3.5. Use Content Security Policy (CSP) Headers

- Implement CSP headers to restrict the types of content that can be loaded and executed by the client, enhancing security against XSS.
- In case of a CSP violations, they will be reported to an endpoint we specify. `/csp-report-endpoint`.

```
import org.springframework.security.config.annotation.web.builders.HttpSecurity;
import org.springframework.security.config.annotation.web.configuration.EnableWebSecurity;
import org.springframework.security.config.annotation.web.configuration.WebSecurityConfigurerAdapter;

@EnableWebSecurity
public class SecurityConfig extends WebSecurityConfigurerAdapter {

    @Override
    protected void configure(HttpSecurity http) throws Exception {
        http
            .headers()
                .contentSecurityPolicy("default-src 'self'; script-src 'self'; object-src 'none'; style-src 'self' 'unsafe-inline'; img-src 'self' data:; font-src 'self' data:; report-uri /csp-report-endpoint")
                .and()
            .and()
            .authorizeRequests()
                .anyRequest().authenticated();
    }
}

```

Header name	Description

`default-src 'self';`	Restricts loading content like JavaScript, images, CSS, fonts, etc... to the same origin (`'self'`).
`script-src 'self';`	Only allows JavaScript to be loaded from the same origin.
`object-src 'none';`	Disallows the loading of any plugins or objects (like Flash or Java applets).
`style-src 'self' 'unsafe-inline';`	Restricts the loading of CSS to the same origin. The `'unsafe-inline'` allows inline styles
`img-src 'self' data:;`	Limits image sources to the same origin and allows data URIs for images.
`font-src 'self' data:`	Limits font sources to the same origin and allows data URIs for fonts.

---

4. Authentication and Authorization

4.1. Authentication

- Always adhere to the accepted authentication mechanism for the application.
- Consider token-based authentication for added security.
- Enforce strong password policies such as minimum length, complexity, and expiration. Refer Password Management
- Implement account lockout mechanisms after a defined number of failed login attempts to prevent brute force attacks.
- Enforce account disabling after five invalid login attempts. The account must be disabled for a period of time sufficient to discourage brute force guessing of credentials, but not so long as to allow for a denial-of-service attack to be performed.
- Re-authenticate users prior to performing critical operations. If long authenticated sessions are allowed, periodically re-validate a user's authorization to ensure that their privileges have not changed and if they have, log the user out and force them to re-authenticate. This is not

implemented as of now. The critical operations should be determined and it should be coordinated with the business requirements as well. The development effort would be at Medium level.

- Utilize authentication for connections to external systems that involve sensitive information or functions. Authentication credentials must be protected by following configuration management guidelines.
- Use only HTTP POST requests to transmit authentication credentials.
- Use Spring Security inbuilt features to manage authentication for all pages and resources, except those specifically intended to be public.

> **Spring Security:** Provides authentication and authorization mechanisms, protecting your API from unauthorized access.
>
> - Authentication and authorization RBAC-with-Spring-Security
> - Session management Session-Management-Frameworks
> - CSRF protection Session-Management-Frameworks
> - Input validation Java Coding Standards | 7. Input Validations

- Implement monitoring to identify attacks against multiple user accounts, utilizing the same password. This is not implemented as of now. Proper logs and metrics should be configured to create alerts.

> **Logging:** Add logs to monitor the username and hashed password
>
> **DataDog:** Add a metric to create an alert based on hashed password occurrences

4.2. Authorization

4.2.1. Role-Based Access Control (RBAC)

Ensure users have access only to the resources they are authorized to use based on the roles assigned. Users and services should only have the minimum permissions necessary to perform their tasks. This is the **Principle of least privilege**. RBAC is achieved by permissions being assigned to roles rather than individuals, and roles are assigned to users.

- Use a fine-grained roles and responsibility matrix to restrict access to specific data elements or functionalities. Roles and Responsibilities Matrix
- Ensure no single user has control over all aspects of a critical task.
- Ensure that in the absence of explicit permissions, access is denied by default.

4.2.2. RBAC with Spring Security

Use roles and authorities(privileges) clearly and ensure that access to endpoints, and services is controlled based on these roles.

Endpoint Level (HTTP Security)

- Enforce authorisation controls on every request.
- Restrict access to protected URLs to only authorised users

```
@Override
protected void configure(HttpSecurity http) throws Exception {
    http
        .authorizeRequests()
            .antMatchers("/admin/**").hasRole("ADMIN")
            .antMatchers("/user/**").hasAnyRole("USER", "ADMIN")
            .antMatchers("/public/**").permitAll()
            .and()
        .formLogin()
            .permitAll()
            .and()
        .logout()
            .permitAll();
}
```

Class Level (Service Layer Security)

Restrict access to services to only authorised users. Apply the `@Secured` annotation at the class level, which will secure all methods within the class.

```
@Service
@Secured("ROLE_ADMIN")
public class AdminService {

    public void performAdminTask() {
        // This method is accessible only to users with ROLE_ADMIN
    }

    public void anotherAdminTask() {
        // This method is also accessible only to users with ROLE_ADMIN
    }
}

```

Method Level (Service Layer Security)

Restrict access to protected functions in specific service methods to only authorised users by using annotations `@PreAuthorize`, `@Secured`, or `@PostAuthorize`.

```
@Service
public class UserService {

    @PreAuthorize("hasAuthority('READ_PRIVILEGES')")
    public void readData() {
```

```
        // method implementation
    }

    @PreAuthorize("hasAuthority('WRITE_PRIVILEGES')")
    public void writeData() {
        // method implementation
    }
}

```

Exceptions occur in the above scenarios must be handled using Global exception handling. Refer Java Coding Standards | 6. Exception Handling

UI Level (Frontend Security)

In a Thymeleaf template, use `sec:authorize` to conditionally display content based on roles. The front end rules for authorization, must match with server side authorization rules.

```
<div sec:authorize="hasRole('ADMIN')">
    <!-- Admin-specific content -->
</div>
<div sec:authorize="hasRole('USER')">
    <!-- User-specific content -->
</div>
```

It is recommended to use a **centralized code library** for all authentication and authorization controls. Segregate authentication logic from the resource being requested and use redirection to and from the centralized authentication control.

4.3. Multi-Factor Authentication (MFA) (TBC)

- All users with administrative privileges must have enabled MFA.
- Use the most suitable MFA Option out of SMS, email, authenticator app, hardware token...etc
- Use MFA for highly sensitive or high value transactional accounts.
- Currently, there is no MFA process followed. Selection of the most suitable MFA solution is still in progress.

---

5. Access Control

Access Control Policy should document the platform's business rules, data types and access authorisation criteria and/or processes so that access can be properly provisioned and controlled. This includes identifying access requirements for both the data and system resources.

5.1. Access Control Validation

- Deny all access if the application cannot access its security configuration information.
- Implement appropriate access controls for sensitive data stored on the server. This includes cached data, temporary files and data that should be accessible only by specific system users
- **OWASP ZAP** (Zed Attack Proxy) can be integrated into **Jenkins CI/CD pipeline** to test for access control vulnerabilities. This setup helps ensure that security scans are performed consistently, and vulnerabilities are detected early in the development process. Infrastructure effort and development effort would be significant for the initial setup.

5.2. API Security

- **API Authentication**: Use OAuth2 or other secure token-based authentication mechanisms for API access.
- **Rate Limiting**: Implement rate limiting to prevent abuse of API endpoints.

5.3. Access Reviews and Audits

- Conduct periodic reviews of user access rights, especially for sensitive systems and data. Ensure that only authorized personnel have access to critical systems, and also ensure they align with job functions.
  - **Quarterly Review:** Conduct a comprehensive review of access permissions every three months.
  - **Event-Driven Review:** Review access permissions immediately after significant organizational changes, such as changes in team structure, role changes, or employee off-boarding.
  - **Annual Review:** Perform a more thorough review annually, which includes validating that all permissions are appropriate, removing obsolete accounts, and ensuring that least privilege principles are maintained.
- Implement account auditing and enforce the disabling of unused accounts.

---

6. Password Management

Refer Password Policy and Procedures for detailed information.

---

7. Session Management

Session data is important in authentication, authorization and state management in a web application. Protecting session data from unauthorized access and ensuring session integrity is

crucial to prevent attacks like session hijacking, session fixation and CSRF. (Cross Site Request-Forgery). **Spring Security** manages the complete session life-cycle, including creating, invalidating, and regenerating session IDs as needed (e.g., after a user logs in or logs out). Spring Security is used in all the projects as per the current implementation.

7.1. Secure Session IDs

- By default, Spring Security creates and manages the session ID securely by servlet container using a cryptographically secure random number generator.
- It is stored in a session cookie that is marked as `HttpOnly` and `Secure`, reducing the risk of it being stolen via client-side scripts.

7.2. Session Expiry

- Session timeout for the application is configured in environment based property file.

```
server.servlet.session.timeout: 30m  # Set session timeout to 30 minutes
```

- When a user logs out, Spring Security automatically invalidates the session.

```
@Override
protected void configure(HttpSecurity http) throws Exception {
    http
        .logout()
            .invalidateHttpSession(true) // Invalidate the session on logout
            .deleteCookies("JSESSIONID") // Optionally delete the session cookie
            .logoutSuccessUrl("/login?logout") // Redirect to login page after logout
            .permitAll();
}
```

- When the server is restarted, all sessions are typically invalidated.
- Session can be invalidated programmatically.

```
public String invalidateSession(HttpServletRequest request) {
        HttpSession session = request.getSession(false);
        if (session != null) {
            session.invalidate(); // Invalidate the session
        }
        return "session invalidated";
    }
}
```

7.3. Session Storage

- **In-Memory Storage:**
  - By default, session data is stored in memory on the server side, associated with the session ID.
- **Session Attributes**
  - Spring Security stores authentication details such as user details, username in the session.

- It does not store sensitive information like passwords. Instead, only necessary authentication and user-related data such as authorities are kept in the session.
- The session is also protected from unauthorized access by associating it with the authenticated user.

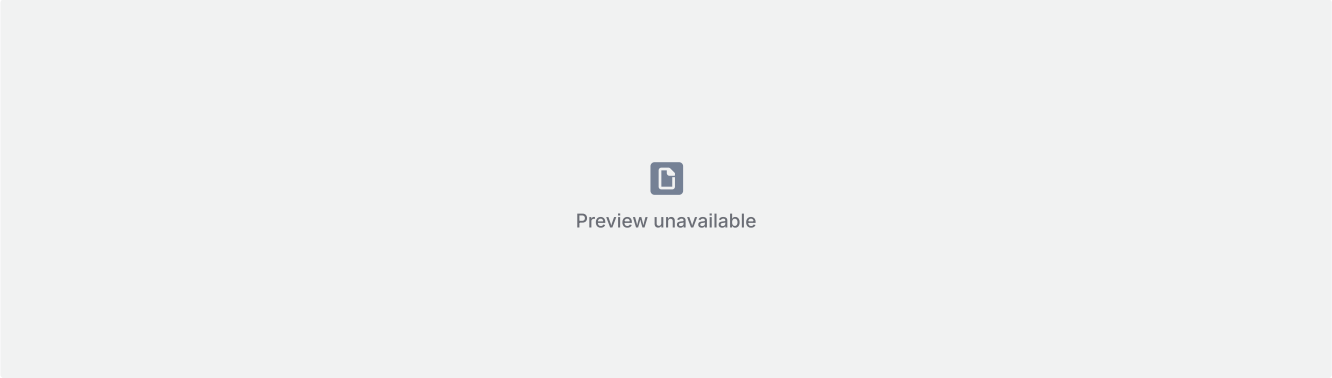
- **Redis**
  - As per the current setup, Redis is utilized as a caching mechanism as well as session storage.
  - Redis server is not publicly accessible.
  - TLS is enabled for Redis server.

7.4. Cookies

- SAE Web is an ‘Open web application’. It uses a cookie to identify the web user and his session.
- SAE Hub is a ‘Private web application’. All its session related data is managed using Spring Security.
- The BFF uses cookies to create a user session, which is directly associated with the user's tokens, either through server-side or client-side session state. Given the sensitive nature of these cookies, they must be properly protected.

The following cookie security guidelines are relevant for this particular BFF architecture:

- The BFF MUST enable the *Secure* flag for its cookies.
- The BFF MUST enable the *HttpOnly* flag for its cookies.
- The BFF SHOULD enable the *SameSite=Strict* flag for its cookies. But if the web page uses analytics services, embedded resources and iframes, this could be set to *SameSite=None.*
- The BFF SHOULD set its cookie path to /
- The BFF SHOULD NOT set the *Domain* attribute for cookies. But the `Domain` attribute of a cookie can be set, when we need the cookie to be accessible across **subdomains** of a particular domain.
- The BFF SHOULD start the name of its cookies with the *__Host-* prefix.



Refer OAuth 2.0 for Browser-Based Applications for further information.

---

8. Data Protection

8.1. Minimize Data Collection

- Collect only the data that is strictly necessary for the application's functionality. Specify and document the purposes for which personal data is collected and processed.
- Card payment related data

8.2. Right to be Forgotten

- The application should support the removal of sensitive data when that data is no longer required. (e.g. personal information or certain financial data).
- Provide functionality to delete user accounts and related personal data upon request. Currently some applications supports this up to some extent. Some applications can restrain data from being displayed in the application, while retaining data in the database.

```
@Service
public class UserService {

    @Autowired
    private UserRepository userRepository;

    public void deleteUser(Long userId) {
        userRepository.deleteById(userId);
    }
}

```

8.3. System Information

- Remove all comments in user accessible production code that may reveal back-end system or other sensitive information
- Remove unnecessary application and system documentation as this can reveal useful information to attackers

- Do not include sensitive information in HTTP GET request parameters.
  - Use post requests if sensitive information must be sent in the request
  - Use tokens or references rather than using sensitive information directly.

---

9. Error Handling

9.1. Avoid Exposing Sensitive Information

- Never expose sensitive and Personally identifiable information (PII) data such as cardholder data, passwords, Passport numbers in error messages to the end-user.
- Authentication failure responses should not indicate which part of the authentication data was incorrect. For example, instead of "Invalid username" or "Invalid password", just use "Invalid username and/or password" for both.

9.2. Generic Error Messages

- Provide generic error messages to the end-user without revealing sensitive details.
- Do not expose stack traces, debugging information, sensitive information, or detailed error messages to the end-users.
- Do not expose system details, session identifiers or account information or logs.
- Instead, provide user-friendly error messages that do not reveal internal details.

> Defining common Error message Template for error message categories, would be highly beneficial. Every error message in the system should be adhered to a template. Development effort for this would be significant, and highly dependable with business expectations.

9.3. Centralized Error Handling

Use Spring Boot provided annotations like `@ControllerAdvice` and `@ExceptionHandler` to handle exceptions and return custom error responses. This helps in returning consistent and secure error responses.

Java Coding Standards | 6. Exception Handling

9.4. HTTP Status Codes

Use appropriate HTTP status codes to indicate error conditions.

API Conventions and Guidelines | 4.2.1. HTTP Response Codes

9.5. Custom Error pages

Create custom error pages for common HTTP status codes like 404, 500 to prevent default server error pages from exposing server information. Use `@ControllerAdvice` class to handle exceptions and return custom error responses.

---

10. Logging

10.1. Sensitive and PII data

- Never log sensitive information such as passwords, security codes, or any other PII (Personally Identifiable Information).

PII data

Data	Application	Saved in DB	Logged in files	Data Redaction Applied
Full Name	All	✅	✅	None
Date of Birth	All	✅	✅	None
Passport Number	Expat	✅	❌	None
Address (Street, City, State, Postcode)	All	✅	✅	None
Email Address	All	✅	✅	None
Phone Number	All	✅	✅	None
Last 4 digits of Credit Card Number	All	✅	❌	Truncation
Card holder name	All	✅	❌	-
Expiration date of Credit Card Number	All	✅	❌	-
Medical related information	All	✅	✅	None
Claims related information	All	✅	❌	None

- Mask or redact sensitive information if logging is unavoidable.

- **Truncating:** Log only the last four digits of a credit card number
- **Mask Names:** Mask all but the first letter of the first and last names. `John Doe` → `j*** D**`
- **Mask Email address:** Mask the username part of the email, leaving the domain visible. `john.doe@example.com` → `j***.d**@example.com`
- **Mask phone numbers:** Mask the middle digits of the phone number, leaving the country code and last few digits visible. `+44 117 2345678-> +44 117 ****678`

- Only log the minimum amount of information needed for debugging or auditing purposes.

10.2. Log Rotation

Current log rotation policy which is `TimeBasedRollingPolicy` is implemented using **Logback** in all applications, to manage log file size and prevent disk exhaustion, and also to simplify the debugging process.

```
<appender name="HIKARI_APPENDER" class="ch.qos.logback.core.rolling.RollingFileAppender">
        <File>${CATALINA_HOME}/logs/hikari/hikari.log</File>
        <encoder>
            <Pattern>%(%d{MM-dd HH:mm:ss}) - %msg%n</Pattern>
        </encoder>
        <rollingPolicy class="ch.qos.logback.core.rolling.TimeBasedRollingPolicy">
            <FileNamePattern>${CATALINA_HOME}/logs/hikari/hikari.%d{yyyy-MM-
dd}.log</FileNamePattern>
        </rollingPolicy>
    </appender>
```

10.3. Log Encryption

- Encrypting log files at rest must be enforced, if storing sensitive information is unavoidable in servers.
- However, as all the application log files are now in **Cloudwatch Logs**, automatic encryption is applied on log files at rest.
- Amazon CloudWatch Logs automatically encrypts log data at rest using AWS-managed encryption keys by default.

10.4. Log Retention and Monitoring

- Implement log retention policies that align with PCI DSS requirements, ensuring that logs are retained for the required period and securely archived.
- Monitor logs for suspicious activity and configure alerts for potential security breaches.

> **Rapid7 FIM:** Rapid7 File Integrity Monitoring (FIM) is a security feature designed to detect and monitor changes to critical files and directories on a system. It is part of Rapid7's

broader security solutions, which include vulnerability management, incident detection, and response capabilities.

10.5. Intrusion Detection

Use CloudWatch Logs to detect potential security incidents. Create Metric Filters to generate CloudWatch metrics from the log data. Filters could be created to detect:

- **Failed login attempts**: Track authentication failures.
- **Unusual API calls**: Monitor for unexpected API requests.
- **Security group changes**: Identify changes to security configurations.

To identify these, all authentication failures, API request failures must be properly logged at the source code level.

10.6. Logging Level

- Configure logging levels appropriately to balance detailed logging for troubleshooting and security with performance and storage considerations.
- Use appropriate logging levels (e.g., DEBUG, INFO, WARN, ERROR) and ensure that error-level logs are monitored for signs of system or application issues.
- Currently enabled log level in production environments is INFO. DEBUG Logs can be enabled with a configuration setting, on if necessary, as this can expose internal application logic.
- Ensure that logs include enough detail to identify the source of an issue, such as timestamps, IP addresses, and user IDs, but without including sensitive data.

10.7. Log Aggregation and Analysis

Use a log aggregation tool to centralize log management and analysis. This solution can provide real-time analysis and alerts which aids in monitoring and securing logs.

DataDog: Effectively integrate logs from multiple sources, providing centralized logging, real-time monitoring, and powerful analytics

10.8. Audit Logging

Implement audit logging to track changes to sensitive data and configurations, ensuring compliance with PCI requirements for traceability and accountability.

Spring Security Auditing: Provides auditing capabilities that can be integrated with Spring Data JPA to log and audit data changes, which is crucial for compliance. Development effort would be Medium level.

Audit4j: A lightweight auditing framework for Java applications that can be used to log and audit security-relevant events.

10.9. Logging information

Failures, errors and exceptions are logged in all the systems as per the current implementations. But the back-end logging may not be adhering to a one standard, on log levels and information in the log messages. Front end error messages are following the business requirements.

- Log all security-relevant events, such as authentication failures, access control violations, and suspicious activities.
- Log all input validation failures related to authentication
- Log all authentication attempts, especially failures
- Log all access control failures
- Log all apparent tampering events, including unexpected changes to state data
- Log attempts to connect with invalid or expired session tokens
- Log all system exceptions
- Log all administrative functions, including changes to the security configuration settings
- Log all back end TLS connection failures
- Log cryptographic module failures

Rectifying all the aspects relevant to logging in source code, would need High level of development effort.

---

11. Cryptographic Practices

Refer System Cryptographic Practices for detailed information.

---

12. Communication Security

- Application servers are properly configured with SSL/TLS certificates during server setups, so that they will handle HTTPS connections securely.
- Current Java applications use **WebClient** library, for making HTTPS requests to other applications. This library uses the JVM's default trust store that includes common root certificates to validate SSL certificates of the servers during the communications.
- If the Redis server is hosted in a cloud environment or accessed over a public network, enabling SSL is crucial to protect data in transit.

- As per the current implementation, SSL is enabled on the Redis server, and Redis Client had been configured in the application to use SSL.

```
    @Bean
    public LettuceConnectionFactory lettuceConnectionFactory() {
        RedisClusterConfiguration clusterConfiguration = new
RedisClusterConfiguration(clusterNodes);
        clusterConfiguration.setPassword(RedisPassword.of(password));
        LettuceClientConfiguration.LettuceClientConfigurationBuilder builder =
LettuceClientConfiguration.builder();
        builder.useSsl().and()
                .clientOptions(ClientOptions.builder()
                        .socketOptions(SocketOptions.builder().keepAlive(true).build())
                        .build());
        LettuceClientConfiguration lettuceClientConfiguration = builder.build();
        return new LettuceConnectionFactory(clusterConfiguration,
lettuceClientConfiguration);
    }
```

---

13. Configuration Management

13.1. Environment-Specific Configurations

- Current implementation of configuration management is using environment-specific configuration files to differentiate settings between environments. (.property and .yaml files)
- Use profiles or features to activate environment-specific configurations for QA, UAT and PRODUCTION.

13.2. Configuration As Code

- To Access environment variables specified in the configuration files, use `System.getenv()` or Spring's `@Value` annotation.

```
    @Value("${web-quote-bff.api-key}")
    private String addressServiceAuthKey;

    @Value("${web-quote-bff.api-endpoint}")
    private String addressServiceEndPoint;
```

- Use `@ConfigurationProperties` annotation, to load configuration properties from an external file to a Java file. This is beneficial for hierarchical structures of configuration data.
- Use `@ConditionalOnProperty` annotation to conditionally configure beans based on the presence or value of specific properties in the application configuration files.

13.3. Secret Management and Encryption

- Never hard-code secrets such as API keys, passwords or tokens in code. Never provide examples for the same parameters.
- Encrypt sensitive configuration data at rest and in transit.
- Use a dedicated secret management tool like **AWS KMS, HashiCorp Vault** or **AWS Secrets Manager**. They provide robust features for storing, retrieving, and managing secrets in a secure manner.
- As of current implementations, there is no use of key-management, or secret storage except environment specific configuration files.

Suggested Options

- **AWS Secrets Manager:** Securely manage, rotate secrets such as database credentials, API keys, passwords, and other sensitive information needed by applications.
- **Leverage AWS KMS:** Use AWS Key Management Service (KMS) to manage and rotate encryption keys.

AWS SDK for Java offers functionality to interact with **AWS Secrets Manager**.

- Seamless integration with AWS services.
- Supports different secret types (secrets, parameters, and AWS credentials).
- Offers rotation, versioning, and access control features.

13.4. Configuration Validation

- Implement validation to ensure configuration values are correct and consistent. Use a JSON Schema definitions to validate configurations. Libraries such as Jackson DataFormat, JsonSchemaValidator provide functions to compare the configuration against the schema and return validation results.
- Provide default values for optional configurations.
- Use unit tests to validate configurations

13.5. Configuration Hierarchy

- Define a clear configuration hierarchy with default values, environment-specific overrides, and application-specific settings.
- If there are multiple configuration sources like system properties, environment variables, configuration files, prioritize them to avoid conflicts.

```
mo:
    eligibility:
        travellingFrom:
```

```
            expat:
                enable: true
            other:
                enable: true
    organiserDetails:
        country:
            question:
                enable: false
        email:
            mandatory:
                enable: false
        marketing:
            email:
                enable: true
            post:
                enable: true
            sms:
                enable: true
            telephone:
                enable: true
```

13.6. Configuration Auditing

- Log configuration changes and access.
- Implement auditing mechanisms to track configuration modifications.
- As per the current Master Data Management flow, auditing is not available for every section. Development effort would be as Medium level.

13.7. Secure Default Configurations

- Use secure default configurations whenever possible.
- Avoid overly permissive default settings.

> Spring Cloud Config Server could be setup to externalize and centralize all application configurations used the source code. These configurations can be managed more securely.
>
> The development effort would be significant in terms of configuration migration and testing all the applications' functionalities.

---

14. Database Security

14.1. Data Access

- Use **Spring Data JPA** framework features. Use method query derivation or named queries to prevent injection issues. Always use parameterized queries or named parameters within native queries.
- Spring Data JPA's method naming conventions allow you to create query methods without writing explicit JPQL. This reduces the chances of introducing vulnerabilities through custom SQL queries.

```
public interface UserRepository extends JpaRepository<User, Long> {
    User findByUsername(String username);
}
```

14.2. Entity Validation

- Leverage **Hibernate Validator** for data validation at the entity level for data formats and constraints. Refer Java Coding Standards | 7.2. Entity Validations

14.3. Configuration

- Database credentials and connection strings should not be hard coded anywhere within the application.
- Configurations should always be stored in the accepted Configuration Management system.
- Currently, database credentials are stored in environment specific property files separately for DEV, QA and PRODUCTION servers.

14.4. Access and Permissions

- Grant database users only the necessary permissions to perform their tasks. Avoid granting excessive privileges. The application should use the lowest possible level of privilege when accessing the database.
- Use secure credentials for database access. Remove or change all default database administrative passwords. Utilize strong passwords/phrases or implement multi-factor authentication.
- The application should connect to the database with different credentials for every trust distinction (e.g., user, read-only user, guest, administrators)

14.5. Database Setup

- Disable all unnecessary database functionality such as stored procedures or services, utility packages. Install only the minimum set of features and options required.

- Remove unnecessary default vendor content such as sample schema. Disable any default accounts that are not required to support business requirements.
- Use built-in encryption capabilities of the database to encrypt data at rest.

---

15. File Management

Improper handling of files, specially with sensitive data and information can lead to data breaches, compliance failures, and operational issues.

15.1. Data in Files

- Validation Certificates, Medical Certificates and Claims Documents are generated on-demand using HTML templates, using relevant data from the database. They are not pre-processed and stored in anywhere. Therefore, these documents do not contain user specific sensitive data.
- Files must be encrypted if they contain sensitive data. **JCA** should be used to implement **AES-256 encryption** for files. As per the current implementation, Medical XML files are encrypted using AES-256 and stored in Amazon S3 location.

15.2. Temporary Files

If temporary files are necessary, use a secure, temporary directory (e.g., `/tmp`) and delete them promptly after use.

15.3. File Permissions

- Restrict file permissions to the minimum required for the application to function. Access controls are in place for AWS S3 buckets.
- Require authentication before allowing a file to be uploaded. Medical XMl s are uploaded to s3 location always through application.

15.4. File Naming Conventions

- Use clear and descriptive file naming conventions to aid in organization and management.
- ‘Policy Documents’ are static documents provided by the business.
- HTML templates(without any data) for validation certificates, medical documents, and claims documents are stored in the source code itself. There is a folder structure based on the product and date.

15.5. Error Handling and Logging

- Graceful error handling for file operations must be implemented, including exceptions, file not found, access denied, no such file, invalid file format, unsupported encoding. Java Coding

Standards | 6. Exception Handling principles must be followed in these scenarios. These exceptions are already being properly handled in the applications.

- Log important file-related activities using proper log levels. Avoid logging sensitive data.

15.6. Validation of uploading files

- For file uploads, validate file types, sizes, and content to prevent malicious uploads.
- Uploaded files should only be that are needed for business purposes.
- Uploaded files must include the expected file headers. Checking for file type by extension alone is not sufficient.
- Uploading of any executable file such as .php, .asp, .jsp, .py, .js must be restricted.
- When referencing existing files, use a white list of allowed file names and types. Use regex patterns to allow only accepted file names and types.
- Validate the value of the parameter being passed and if it does not match one of the expected values, either reject it or use a hard coded default file value for the content instead.

15.7. File Storage

Storing files in the web context can open the application to path traversal and injection attacks, where malicious users can attempt to access or modify files outside of the intended directory.

- Files should either go to the content server like **Amazon S3 or in the database** for smaller files or metadata associated with files.
- Absolute file paths should never be sent to the client.
- If input data should be used into a dynamic redirect, then the redirect should accept only validated, relative path URLs.
- Do not pass directory or file paths, use index values mapped to pre-defined list of paths. Directly referencing file paths can introduce security risks and make code maintenance challenging. Using index-based referencing provides a more secure and flexible approach.

```java
import java.util.Map;

@Configuration
public class FilePathConfig {

    @Value("${file.paths}")
    private Map<Integer, String> filePaths;

    public String getFilePath(int index) {
        return filePaths.getOrDefault(index, ""); // Handle missing index gracefully
    }
}

@Service
public class PaymentService {
```

```
16
17      @Autowired
18      private FilePathConfig filePathConfig;
19
20      public void processPayment(PaymentRequest paymentRequest) {
21          // ... payment processing logic ...
22
23          // Accessing a configuration file
24          String configFilePath = filePathConfig.getFilePath(10); // Assuming configuration
    file is at index 10
25          // Load configuration from configFilePath
26
27          // Logging a transaction
28          String logFilePath = filePathConfig.getFilePath(20); // Assuming log file is at
    index 20
29          // Write transaction details to logFilePath
30      }
31  }
```

Asynchronous Processing: Consider asynchronous processing for large file operations to improve performance and responsiveness.

---

16. Memory Management

Memory management is crucial for ensuring that the web application is efficient, responsive, and scalable. Inefficient memory usage can introduce vulnerabilities, and some security measures can impact memory consumption as well. Common Memory Concerns of any application;

- **Heap Space:** Insufficient heap space can lead to `OutOfMemoryError`.
- **Garbage Collection:** Frequent or long garbage collection pauses can impact performance.
- **Memory Leaks:** Objects not being released when no longer needed can consume memory over time.
- **Large Objects:** Creating large objects frequently can increase GC overhead.
- **Database Connections:** Unmanaged database connections can lead to resource exhaustion.

Best Practices for secure coding in terms in memory management are as follows.

- **Object Reuse:** Reuse objects when possible to reduce object creation overhead. Use DTOs for read-only data retrieval

- **String Handling:** Be mindful of string concatenation and usage of `StringBuilder` or `StringBuffer`. Avoid string concatenation using `+`.
- **Collections:** Choose appropriate collections based on data size and access patterns. Implement bounds checks to prevent array index out-of-bounds exceptions.
- **Lazy Loading:** Load data only when needed to avoid unnecessary object creation.
- **Avoid Unnecessary Object Creation:** Minimize object creation within loops or hot code paths.
- **Efficient Algorithms:** Use efficient algorithms and data structures.
- **Optimize Hibernate and JPA:** Batch fetch related entities to reduce the number of queries.
- **Limit Data Retrieval**: Pagination or limit query results to avoid loading large datasets into memory.

> **Apache Commons Lang:** Offers utility classes for string manipulation, encoding, and other common tasks, helping to prevent common vulnerabilities.
>
> **Hibernate Validator:** Validates Java objects based on annotations, ensuring data integrity and preventing injection attacks.
>
> Utilizing these libraries could provide global guidelines on the source code.

> **Spring Boot Actuator:** Provides endpoints to monitor various aspects of your application, including memory usage, garbage collection, and heap dumps. These analysis can be used to fine tune the heap sizes and GC settings.

---

17. Dependency Management

- All dependencies (libraries, frameworks, etc.) must be kept up-to-date.
- Always declare the versions of your dependencies explicitly in the `build.gradle` file.

```
dependencies {
    implementation("org.springframework.boot:spring-boot-starter-web:3.0.0")
}
```

- Dependencies must be regularly scanned for vulnerabilities using automated tools. **JFrog Xray** has been integrated into Jenkins pipelines to identify issues with libraries or transitive dependencies.

- Ensure that dependencies are obtained from trusted sources.
- Leverage the `io.spring.dependency-management` plugin to align the dependencies with the versions provided by Spring Boot's BOM (Bill of Materials).

```
plugins {
    id 'io.spring.dependency-management' version '1.1.0'
}

dependencyManagement {
    imports {
        mavenBom "org.springframework.boot:spring-boot-dependencies:3.1.0"
    }
}
```

- Avoid including unnecessary transitive dependencies by marking them as `implementation` instead of `compile`.

```
dependencies {
    implementation 'com.fasterxml.jackson.core:jackson-databind:2.15.0'
    implementation 'org.apache.commons:commons-lang3:3.12.0'
}
```

---

18. General Practices

18.1. Synk

- A static code analysis tool that identifies potential security flaws directly in the code. It detects vulnerabilities, providing insights and remediation guidance.
- Developers must make sure that Snyk security scan is passed before every commit.
- Snyk plugins should always be up to date.

Refer Snyk - Security Testing by design

18.2. Troubleshooting

- Production log files must never be downloaded into developer machines.
- Use Cloudwatch Logs, DataDog dashboards and consoles to view and debug the flows.
