payara®

Zero Trust Architecture with Jakarta EE and MicroProfile

Developer Guide

User Guide

Contents

Guide Updated: **January 2026**

Introduction

Zero Trust security operates on a fundamental principle: never trust, always verify. This guide demonstrates how to implement Zero Trust architecture using standard Jakarta EE 11 and MicroProfile 6.1 APIs, without resorting to custom security code or vendor-specific extensions. The sample healthcare application developed for this guide serves as a practical example, but the patterns and techniques shown here apply to any enterprise application requiring defense-in-depth security.

Traditional perimeter-based security assumes that once a request passes the firewall, it can be trusted. Zero Trust rejects this assumption. Every request, regardless of origin, must prove its identity, demonstrate authorization, and pass through multiple verification layers before accessing protected resources. Jakarta EE and MicroProfile provide the standardized APIs needed to implement this architecture in a portable, maintainable way.

Zero Trust Principles in Jakarta EE

Zero Trust architecture rests on three core principles, each of which maps directly to Jakarta EE and MicroProfile capabilities. First, verify explicitly: always authenticate and authorize based on all available data points. Jakarta Security's IdentityStore and MicroProfile JWT can handle identity verification, while Jakarta Security's authorization annotations can be used to enforce access control.

Second, use least privilege access: limit user access with just-in-time and just-enough permissions. Jakarta Security's @RolesAllowed annotation provides coarse-grained control, while custom interceptors can be implemented to enable fine-grained attribute-based access control. The combination ensures users receive only the permissions they need for their current task.

Third, assume breach: minimize blast radius and verify end-to-end encryption. Jakarta Interceptors can be used to enable comprehensive audit logging, capturing who accessed what data and when. Jakarta Bean Validation can be used to prevent malformed data from entering an application. Custom then annotations enforce encryption requirements for sensitive operations.

These principles work together to create defense in depth. If an attacker compromises one layer—perhaps stealing a valid JWT token—they still face authorization checks, audit logging, rate limiting, and encryption requirements. Each layer operates independently, preventing single points of failure.

Zero Trust Principle	Jakarta EE/MicroProfile Capabilities
Verify Explicitly	Jakarta Security's `IdentityStore` and MicroProfile JWT for identity; Jakarta Security's authorization annotations for access control.
Use Least Privilege Access	Jakarta Security's `@RolesAllowed` for coarse-grained control; custom interceptors for fine-grained attribute-based access control.
Assume Breach	Jakarta Interceptors for comprehensive audit logging; Jakarta Bean Validation for data integrity; custom annotations for encryption.

Authentication with Jakarta Security and OAuth2/OIDC

Jakarta Security provides the `IdentityStore` interface as a bridge between external identity providers and the application security context. Rather than implementing password verification, account management, or credential storage, an application can delegate these responsibilities to specialized security systems like Keycloak.

The `IdentityStore` implementation can then act as an adapter, translating between OAuth2/OIDC protocols and Jakarta Security's standard interfaces. When a user submits credentials, the identity store forwards them to Keycloak's token endpoint using the OAuth2 password grant flow. Keycloak validates the credentials against its user database, applies password policies, checks for account lockouts, and generates a JWT token containing user identity and claims.

This separation of concerns keeps security logic centralized in Keycloak while allowing the application to work with familiar Jakarta Security APIs. The identity store receives the JWT token from Keycloak and extracts roles from its claims structure. Keycloak embeds roles in multiple locations: **realm-level roles** appear in `realm_access.roles`, **client-specific roles** in `resource_access.{client}.roles`, and group memberships in groups. The identity store navigates this structure to build a complete role set.

The `validate` method of the `IdentityStore` interface returns a `CredentialValidationResult` containing the username and extracted roles. Jakarta Security takes this result and establishes a security context for the request. From this point forward, any code in the request can access the authenticated user's identity and roles through the `SecurityContext` interface.

```
@ApplicationScoped
public class KeycloakIdentityStore implements IdentityStore {
    @Inject
    private KeycloakConfig keycloakConfig;
    @Override
    public CredentialValidationResult validate(Credential credential) {
        if (credential instanceof UsernamePasswordCredential) {
            // Call Keycloak token endpoint
            // Extract roles from JWT response
            // Return CredentialValidationResult with roles
        }
        return CredentialValidationResult.NOT_VALIDATED_RESULT;
    }
}
```

This approach demonstrates the core zero trust pattern of delegating authentication to a specialized, hardened identity provider while maintaining standard interfaces in application code. The application never sees passwords, never implements password policies, and gains access to Keycloak's advanced features like multi-factor authentication and anomaly detection. In short, delegate to battle tested systems by not reinventing the wheel.

Token Validation with MicroProfile JWT

Once authenticated, users receive a JWT access token that accompanies subsequent requests. MicroProfile JWT provides automatic token validation without requiring custom JWT parsing or verification code. This eliminates an entire class of security vulnerabilities that arise from incorrect JWT implementation.

MicroProfile JWT operates through configuration rather than code.

Three properties in `microprofile-config.properties` control token validation:

```
mp.jwt.verify.publickey.
location=http://keycloak:8180/
realms/jdd-poland/protocol/
openid-connect/certs

mp.jwt.verify.issuer=http://
localhost:8180/realms/jdd-poland

mp.jwt.verify.audiences=jdd-
healthcare-app
```

The public key location points to Keycloak's JWKS (JSON Web Key Set) endpoint, which publishes the public keys needed to verify JWT signatures. MicroProfile JWT fetches these keys automatically and caches them. When a request arrives with a JWT token, the runtime retrieves the appropriate public key and verifies the token's cryptographic signature.

Beyond signature verification, MicroProfile JWT checks multiple security properties. It verifies the issuer's claim matches the configured value, preventing tokens from other identity providers from being accepted. It checks the audience's claim to ensure the token was issued for this specific application. It validates expiration times, rejecting expired tokens automatically.

After validation succeeds, MicroProfile JWT makes the token available for injection throughout the application:

```
@Inject
private JsonWebToken jwt;

public void processRequest() {
    String username = jwt.getName();
    Set<String> groups = jwt.getGroups();
    Optional<Object> department = jwt.claim("department");
}
```

This declarative approach keeps JWT validation logic in the runtime, not the application. You inject the validated token and access its claims without worrying about signature algorithms, key rotation, or token expiration. The separation between validation (handled by the MicroProfile JWT runtime) and usage (handled by your application code) is a good example of proper security architecture.

The configuration-driven model also enables environment-specific settings. Development environments can point to local Keycloak instances, while production environments use production identity providers. The same application code works in both environments because validation logic lives in configuration, not code.

Authorization - Composing RBAC and ABAC

Authorization in Zero Trust architecture requires multiple layers. Role-Based Access Control (RBAC) provides coarse-grained permissions based on user roles, while Attribute-Based Access Control (ABAC) enables fine-grained decisions based on contextual attributes. Jakarta EE provides standard RBAC through annotations, while interceptors can be used to implement custom ABAC logic.

Declarative Role-Based Access Control

Jakarta Security's @RolesAllowed annotation provides the foundation for authorization. Marking a method with this annotation tells the Jakarta Security runtime to intercept calls and verify the currently executing user possesses the required role. If not, a ForbiddenException is thrown before the method executes.

```
@GET
@Path("/{id}")
@RolesAllowed({"DOCTOR", "NURSE"})
public Response getPatient(@PathParam("id") String id) {
    // Only users with DOCTOR or NURSE role can execute this
}
```

This declarative approach makes authorization requirements explicit and auditable. Security teams can review REST resources and immediately understand access control rules without reading method implementations. The annotation works at both method and class levels, allowing broad rules at the class level with method-level overrides for specific endpoints.

The annotation composes with standard Jakarta REST annotations, creating self-documenting endpoints. Looking at the method signature reveals both its REST contract and security requirements. This composition extends to MicroProfile OpenAPI, which includes role requirements in the generated API documentation.

Jakarta Security evaluates `@RolesAllowed` before method execution, making it an effective defense against unauthorized access. Even if your application code contains bugs, the security check happens first. This separation between security enforcement and business logic prevents security bypasses.

Attribute-Based Access Control with Interceptors

Role-based control proves insufficient for many Zero Trust scenarios. A doctor might have the DOCTOR role but should only access patients in their assigned department. This requires examining attributes beyond simple role membership.

Jakarta Interceptors can be used to enable custom authorization logic through the `@AroundInvoke` mechanism. An authorization interceptor examines the security context, extracts relevant attributes from JWT claims, and compares them against method requirements:

```
@Interceptor
@RequireAttribute(name = "", value = "")
@Priority(Interceptor.Priority.APPLICATION)
public class AuthorizationInterceptor {
    @Inject
    private JsonWebToken jwt;
    @AroundInvoke
    public Object checkAuthorization(InvocationContext context) throws
Exception {

        RequireAttribute annotation = context.getMethod().
getAnnotation(RequireAttribute.class);

        // Extract required attribute from annotation
        // Get user's attribute values from JWT claims
        // Compare and allow/deny access
        return context.proceed();
    }
}
```

The custom `@RequireAttribute` annotation marks the interceptor binding and carries the attribute requirements. Methods can combine `@RolesAllowed` with `@RequireAttribute` to create layered authorization:

```
@GET
@Path("/department/{dept}")
@RolesAllowed("DOCTOR")

@RequireAttribute(name = "department", value = "Cardiology")
public Response getCardiologyPatients() {
    // Requires DOCTOR role AND department=Cardiology claim
}
```

Jakarta Security evaluates `@RolesAllowed` first. If that check passes, the authorization interceptor executes, performing attribute-based validation. This composition creates defense in depth—bypassing one check doesn't grant access.

The interceptor priority system controls execution order. Setting priority to `Interceptor.Priority.APPLICATION` ensures the authorization interceptor runs after authentication but before business logic. This ordering guarantees that authorization decisions have access to the authenticated user's identity and claims.

Attribute-based authorization demonstrates a key Zero Trust pattern: making authorization decisions based on rich context, not just roles. The JWT token carries multiple claims about the user—their department, location, security clearance level—and the interceptor examines whichever claims are relevant for the protected resource.

Audit Logging for Compliance and Forensics

Zero Trust architecture requires comprehensive audit trails. Every access to sensitive data must be logged with sufficient detail to enable forensic analysis and compliance reporting. Jakarta Interceptors provide the mechanism, while Jakarta Persistence can be used to ensure durable storage of audit events.

The audit interceptor wraps method execution, capturing details before, during, and after invocation:

```
@Interceptor
@Audited
@Priority(Interceptor.Priority.APPLICATION + 100)
public class AuditInterceptor {


@Inject
    private SecurityContext securityContext;
    @Inject

    private AuditLogRepository auditLogRepository;

    @AroundInvoke

    @Transactional

    public Object auditMethodCall(InvocationContext context) throws Exception {

        String username = securityContext.getCallerPrincipal().getName();
```

```
        Instant startTime = Instant.now();
        try {
            Object result = context.proceed();
            // Log successful access
            return result;
        } catch (Exception e) {
            // Log failed access with error details
            throw e;
        } finally {
            // Persist audit entry to database
        }
    }
}
```

The `@Transactional` annotation ensures audit logs persist even when the main operation fails. If a database update throws an exception, the audit log still records the attempt. This prevents attackers from hiding their tracks by triggering errors.

The custom `@Audited` annotation carries metadata about the operation being logged:

```
@Target({METHOD, TYPE})

@Retention(RUNTIME)

@InterceptorBinding

public @interface Audited {

    String action() default "";

    SensitivityLevel level() default SensitivityLevel.MEDIUM;

}
```

Sensitivity levels enable prioritization during audit review. CRITICAL-level events—accessing or modifying sensitive data—demand immediate attention when anomalies appear. LOW-level events—statistics queries—provide usage patterns but rarely indicate security issues.

Each audit entry captures rich context: who performed the action (username), from where (IP address), what they attempted (method name and action), when it occurred (timestamp), whether it succeeded (Boolean flag), and how long it took (duration). This context enables sophisticated analysis.

Execution duration proves particularly valuable for detecting anomalies. If accessing a single patient record suddenly takes ten times longer than normal, this might indicate an attacker's dumping data or exploiting an injection vulnerability. Unusual patterns in duration correlate with security events.

The audit repository uses the newly released Jakarta Data for database operations, abstracting SQL while maintaining type safety:

```
@Repository

public interface AuditLogRepository extends DataRepository<AuditLogEntry, UUID>
{

    List<AuditLogEntry> findByUsernameOrderByTimestampDesc(String username);

    List<AuditLogEntry> findBySensitivityLevel(SensitivityLevel level);

}
```

This approach separates audit storage from audit collection. The interceptor focuses on capturing data, while the repository handles persistence. This separation enables changing storage mechanisms—switching from PostgreSQL to a specialized audit database—without modifying the interceptor.

Audit logging demonstrates defense in depth. Even if an attacker gains unauthorized access, their activities leave traces. Forensic analysts can reconstruct what data was accessed, when, and by whom, enabling incident response and compliance reporting.

Enforcing Encrypted Transit

Zero Trust architecture never trusts the network. All sensitive data must transit over encrypted connections, regardless of network topology. Jakarta Interceptors enable enforcement of this requirement through custom annotations.

An encryption interceptor verifies that requests arrive over TLS before allowing access to sensitive endpoints:

```
@Interceptor

@Encrypted

@Priority(Interceptor.Priority.PLATFORM_BEFORE + 200)

public class EncryptionInterceptor {

    @Inject

    private HttpServletRequest request;

    @AroundInvoke

    public Object enforceEncryption(InvocationContext context) throws Exception
{

        Encrypted annotation = context.getMethod().getAnnotation(Encrypted.
class);

        if (annotation.requireTls() && !request.isSecure()) {

            throw new ForbiddenException("HTTPS/TLS required for this
operation");

        }
        return context.proceed();

    }
}
```

The interceptor examines `HttpServletRequest.isSecure()`, which returns true when the request arrives over HTTPS/TLS. Plain HTTP requests to encrypted endpoints fail immediately, before any sensitive data is accessed or transmitted.

Setting the interceptor priority to `PLATFORM_BEFORE + 200` guarantees it runs early in the call chain, before authentication or authorization. This prevents sensitive authentication tokens from being processed over unencrypted connections.

The `@Encrypted` annotation documents encryption requirements directly in code:

```
@GET
@Path("/{id}")
@RolesAllowed({"DOCTOR", "NURSE"})
@Encrypted(algorithm = AES_256_GCM, requireTls = true)
public Response getPatient(@PathParam("id") String id) {
    // Requires HTTPS/TLS
}
```

The algorithm parameter serves as documentation, specifying the expected encryption level. While the current implementation only checks for TLS presence, production systems could extend this to verify TLS version (requiring TLS 1.3) or cipher suite strength (rejecting weak ciphers).

This pattern demonstrates declarative security at the transport layer. Rather than scattering `if (!request.isSecure())` checks throughout code, the annotation makes encryption requirements explicit and enforced. Security teams can audit resources and identify any missing @Encrypted annotations on sensitive endpoints.

Multi-Factor Authentication

> ***Note:*** *Before rolling out your own MFA implementation, check to see if your security provider has it built-in already. For example, Keycloak provides support for MFA out of the box. This section only demonstrates a custom implementation for educational purposes. A core tenet of good security practice is to not reinvent the wheel.*

Password-based authentication provides insufficient security for Zero Trust architectures. Multi-factor authentication adds verification of something the user possesses—typically a device capable of generating or receiving one-time passwords.

The MFA service generates time-based one-time passwords (TOTP) using cryptographically secure random number generation:

```
@ApplicationScoped
public class MfaService {
    private static final SecureRandom RANDOM = new SecureRandom();
    private final Map<String, OtpData> otpStorage = new ConcurrentHashMap<>();
    public String generateOtp(String username, String ipAddress) {
        String otp = generateRandomOtp();
        Instant expiryTime = Instant.now().plus(5, ChronoUnit.MINUTES);
        otpStorage.put(username, new OtpData(otp, expiryTime, false));
        // In production: send via SMS, email, or authenticator app
        return otp;
    }
    public boolean validateOtp(String username, String otp, String ipAddress) {
        OtpData stored = otpStorage.get(username);
        // Check: OTP exists, not used, not expired, matches submitted value
        // Mark as used if validation succeeds
        // Return true/false
    }
}
```

The validation flow enforces multiple security properties. First, an OTP must exist for the user—you can't validate what wasn't generated. Second, the OTP can only be used once; marking it as used prevents replay attacks. Third, OTPs expire after five minutes, limiting the window for interception and use. Finally, the submitted OTP must exactly match the stored value.

Using SecureRandom instead of regular Random ensures OTPs are cryptographically unpredictable. An attacker cannot predict the next OTP by observing previous ones. The six-digit format provides one million possible values, making brute force attacks within the five-minute window infeasible when combined with rate limiting.

The demonstration implementation stores OTPs in memory for simplicity. You should use distributed storage systems like Redis or a similar cache for your production deployment. When multiple application instances run behind a load balancer, OTP validation might occur on a different instance than generation. Distributed storage will ensure all instances see the same OTP data.

Production MFA implementations must never return OTPs in API responses. Instead, send them through secure out-of-band channels: SMS via Twilio or AWS SNS, email via SendGrid or AWS SES, or time-based codes in authenticator apps following the TOTP standard. The sample code returns OTPs only for testing convenience.

MFA demonstrates defense in depth. Even if an attacker steals a valid password—through phishing, database breach, or other means—they cannot authenticate without the second factor. The out-of-band nature of OTP delivery makes coordinated attacks significantly harder.

Rate Limiting and Brute Force Prevention

Zero Trust architecture assumes attackers will attempt authentication with stolen or guessed credentials. Rate limiting prevents brute force attacks by restricting the number of authentication attempts within a time window.

The rate limiter implements a sliding window algorithm using in-memory timestamp tracking:

```
@ApplicationScoped

public class RateLimiter {
    private final Map<String, Queue<Instant>> attemptHistory = new
ConcurrentHashMap<>();
    public boolean isAllowed(String key, int maxAttempts, int windowSeconds) {
```

```
        Queue<Instant> attempts = attemptHistory.computeIfAbsent(key, k -> new
ConcurrentLinkedQueue<>());
        // Remove timestamps outside the window
        Instant cutoff = Instant.now().minusSeconds(windowSeconds);
        attempts.removeIf(timestamp -> timestamp.isBefore(cutoff));
        // Check if under the limit
        if (attempts.size() < maxAttempts) {
            attempts.add(Instant.now());
            return true;
        }

        return false; // Rate limited
    }
}
```

The sliding window approach provides better protection than fixed windows. With fixed windows, an attacker could make maximum attempts at the end of one window and the beginning of the next, effectively doubling their attempts. Sliding windows count attempts within the trailing time period, preventing this bypass.

Different limits apply to different identifiers. IP-based limiting (five attempts per minute) prevents a single source from hammering authentication endpoints. User-based limiting (ten attempts per five minutes) protects against distributed attacks targeting specific accounts from multiple IP addresses.

A scheduled cleanup job prevents unbounded memory growth:

```
@Scheduled(every = "5m")

public void cleanup() {

    rateLimiter.cleanup(maxAgeSeconds);

}
```

The cleanup removes entries with no recent attempts, recovering memory from abandoned attack attempts. Without cleanup, the map would grow indefinitely as attackers tried different IP addresses.

Production deployments require distributed rate limiting. Multiple application instances must share rate limit data; otherwise, an attacker can distribute requests across backend servers to bypass per-instance limits. Redis provides ideal storage for this, with atomic increment operations and TTL-based expiration aligning perfectly with rate limiting requirements.

Rate limiting integrates with the security event system, firing events when limits are exceeded. These events enable real-time alerting and blocking. After repeated rate limit violations from an IP address, the system might add it to a temporary block list or require CAPTCHA completion.

Note on Specialized Rate Limiting Libraries

While this implementation shows rate limiting principles with standard Java collections, for production, you should consider Bucket4j (https://bucket4j.io/) - a specialized rate limiting library that provides:

- **Token bucket and leaky bucket algorithms** - More sophisticated than sliding window approaches
- **Distributed rate limiting** - Native support for Redis, Hazelcast, Infinispan, and other distributed caches
- **Better performance** - Optimized for high-throughput scenarios
- **Flexible configuration** - Multiple rate limit bands (e.g., "100 requests per second AND 10,000 per day")
- **Bandwidth reservations** - Reserve tokens for future use
- **Zero dependencies** - Core library has no external dependencies

Bucket4j integrates well with Jakarta EE applications and provides both synchronous and asynchronous APIs. For production Zero Trust implementations handling high authentication volumes, Bucket4j offers battle-tested rate limiting with predictable performance characteristics.

Example with Bucket4j:

```
Bandwidth limit = Bandwidth.simple(5, Duration.ofMinutes(1));
Bucket bucket = Bucket.builder() .addLimit(limit) .build();

if (bucket.tryConsume(1))
{ // Allow authentication attempt }
else { // Rate limited }
```

Session Management for Stateful Security

While JWT tokens enable stateless authentication, session management adds security capabilities that pure stateless approaches cannot provide. Sessions enable forced logout, detect concurrent access from different locations, and track user activity patterns for anomaly detection.

The session manager in the sample project creates session records when users authenticate:

```
@ApplicationScoped

public class SessionManager {

    private final Map<String, UserSession> activeSessions = new
ConcurrentHashMap<>();

    @Inject

    @ConfigProperty(name = "security.session.timeout", defaultValue =
"1800")

    private Integer sessionTimeoutSeconds;

    public UserSession createSession(String username, String ipAddress,
Map<String, Object> metadata) {

        String sessionId = UUID.randomUUID().toString();

        Instant expiresAt = Instant.now().
plusSeconds(sessionTimeoutSeconds);

        UserSession session = new UserSession(sessionId, username,
ipAddress, Instant.now(), expiresAt, metadata);

        activeSessions.put(sessionId, session);

        return session;

    }

}
```

Sessions support sliding expiration, where activity renews the timeout. This balances security against user convenience—active users don't face unexpected logouts, but abandoned sessions expire quickly. The timeout configuration uses MicroProfile Config, enabling environment-specific values without code changes.

Session validation checks both existence and expiration:

```
public boolean isSessionValid(String sessionId) {
    UserSession session = activeSessions.get(sessionId);
    if (session == null) {
        return false;
    }
    if (Instant.now().isAfter(session.expiresAt())) {
        expireSession(sessionId);
        return false;
    }
    return true;
}
```

A scheduled cleanup job removes expired sessions periodically, preventing the session map from growing unbounded. The cleanup fires expiration events, enabling security monitoring to track session patterns. If a user's sessions frequently expire and immediately recreate from different IP addresses, this might indicate account sharing or compromise.

Session metadata captures context beyond basic authentication. Recording the user agent, IP address, and geographic location enables anomaly detection. If a user typically accesses the system from New York during business hours, then suddenly appears in Moscow at 3 AM, the security system can require additional authentication factors.

Production session management requires distributed storage. Redis excels here, with native support for TTL-based expiration that aligns with session timeout semantics. Multiple application instances share session data through Redis, enabling seamless failover and load balancing.

Security Event System and Monitoring

Zero Trust architecture demands visibility into security operations. Every security-relevant action—authentication, authorization, data access, security violations—must publish events that monitoring systems can observe and analyze.

Jakarta CDI's event mechanism provides the foundation for such a pattern. Security components fire events without knowing who observes them, while monitoring components observe events without knowing their origin. This loose coupling enables adding or modifying monitoring without touching security enforcement code:

```
// Firing events
@Inject
private Event<SecurityEvent> securityEventPublisher;
securityEventPublisher.fire(new SecurityEvent(
    SecurityEvent.Type.AUTHENTICATION_FAILURE,
    username,
    ipAddress,
    Map.of("reason", "Invalid credentials")
));
// Observing events
@ApplicationScoped
public class SecurityMonitor {
    public void onSecurityEvent(@ObservesAsync SecurityEvent event) {
        // Update metrics
```

```
        // Detect anomalies

        // Trigger alerts

    }

}
```

The `@ObservesAsync` annotation ensures event processing happens off the request thread. Monitoring might involve network calls to external systems, database writes for persistent logs, or complex analysis algorithms. Asynchronous observation prevents monitoring overhead from impacting request latency.

Events carry rich context: who performed the action, from where, what type of action, and custom metadata. This context enables sophisticated detection algorithms. The monitoring system tracks patterns for each user—their typical access times, locations, resources accessed—and flags deviations.

Multiple failures from the same username might indicate a brute force attack against that account. Multiple failures from the same IP address might indicate credential stuffing attacks. Successful authentication followed immediately by authorization failures might indicate a compromised account with legitimate credentials but malicious intent.

The event system integrates with all security components. The identity store fires authentication events. Authorization interceptors fire authorization events. The audit interceptor fires data access events. Rate limiters fire suspicious activity events. Each component publishes events independently, yet monitoring observes them holistically.

This architecture demonstrates the Observer pattern at an enterprise scale. Security enforcement and security monitoring remain separate concerns, connected only through well-defined event types. Adding new monitoring capabilities—machine learning-based anomaly detection, integration with SIEM systems—requires only new event observer implementations.

Input Validation with Jakarta Bean Validation

Zero Trust extends to data validation. Never trust client input, even from authenticated and authorized users. Malformed data might indicate bugs in client applications, integration errors, or deliberate injection attacks.

Jakarta Bean Validation provides declarative validation through annotations on domain entities:

```
@Entity

public class Patient {

    @NotBlank(message = "First name is required")

    @Size(min = 2, max = 50)

    private String firstName;

    @Pattern(regexp = "\\d{3}-\\d{2}-\\d{4}", message = "SSN must be in format
XXX-XX-XXXX")

    private String ssn;

    @Email(message = "Invalid email format")

    private String email;

    @Past(message = "Date of birth must be in the past")

    private LocalDate dateOfBirth;

}
```

These annotations serve dual purposes: they document data requirements and enforce them at runtime. Validation occurs automatically when entities are persisted through Jakarta Persistence or when DTOs are submitted to REST endpoints.

Validation prevents multiple attack vectors. Format constraints like @Pattern prevent injection attacks—a Social Security Number matching `\d{3}-\d{2}-\d{4}` cannot contain SQL injection syntax. Size constraints prevent buffer overflows and denial-of-service through oversized data. Business rules like @Past ensure data integrity.

The validation framework integrates with Jakarta REST through exception mapping:

```
@Provider

public class ValidationExceptionMapper implements
ExceptionMapper<ConstraintViolationException> {
```

```
    public Response toResponse(ConstraintViolationException e) {
        List<String> violations = e.getConstraintViolations()
            .stream()
            .map(ConstraintViolation::getMessage)
            .toList();
        return Response.status(400)
            .entity(new ErrorResponse("Validation failed", violations))
            .build();
    }
}
```

This mapper converts constraint violations into meaningful error responses without exposing internal exception details. Clients receive actionable information about validation failures while the system maintains security through controlled error messages.

Validation shows defense in depth. Even if SQL injection prevention through parameterized queries fails, format constraints might block the attack. Even if your application logic contains bugs, validation ensures only well-formed data reaches that logic. Each layer provides independent protection.

Composing Security Layers

The power of this architecture becomes clearer when you compose multiple security layers on a single endpoint. Each annotation adds an independent security control, creating defense in depth without code complexity:

```
@GET
@Path("/{id}")
@RolesAllowed({"DOCTOR", "NURSE"})            // Standard RBAC
@RequireAttribute(name = "department", value = "Cardiology")  // Custom ABAC
@Audited(action = "VIEW_PATIENT", level = CRITICAL)           // Compliance logging
@Encrypted(algorithm = AES_256_GCM, requireTls = true)        // Transport security
public Response getPatient(@PathParam("id") @NotBlank String id) {
    // Business logic executes only after all security checks pass
}
```

Jakarta EE's interceptor priority system controls execution order. Encryption checking happens first, guaranteeing the request arrives over TLS. Authentication verifies identity. Role-based authorization confirms broad permissions. Attribute-based authorization performs fine-grained checks. Input validation ensures parameter correctness. Audit logging captures the access attempt. Finally, business logic executes.

Each layer operates independently. Bypassing one layer doesn't grant access—all must pass. This independence also aids testing and maintenance. Security teams can modify authorization rules without affecting encryption requirements. Compliance teams can adjust audit sensitivity levels without touching authentication.

The declarative approach makes security requirements explicit and auditable. Reading the method signature reveals every security control in play. Security teams don't need to trace through implementation code searching for scattered security checks. Everything is visible at the method declaration.

This composition extends to custom security requirements. Need IP whitelisting? Create an `@RequireIpWhitelist` annotation and interceptor. Need geographic restrictions? Add `@RequireGeolocation`. The pattern remains consistent: define an interceptor binding annotation, implement the interceptor, set the priority, and apply the annotation to protected resources.

Production Considerations

Demonstration code (like the sample project for this guide) and production code differ in several critical areas. The patterns shown here are production-grade, but the infrastructure requires hardening.

In-memory storage for rate limiting, sessions, and MFA state works in single-instance deployments but fails in production clusters. Multiple application instances must share this state, requiring distributed storage like Redis. Redis provides atomic operations for rate limiting, TTL-based expiration for sessions, and pub/sub for cache invalidation across instances.

Secret management requires immediate attention. Never store secrets—client secrets, database passwords, API keys—in configuration files or environment variables. Integrate with HashiCorp Vault, AWS Secrets Manager, Azure Key Vault, or Google Cloud Secret Manager. These systems provide secret rotation, access auditing, and encryption at rest.

MFA delivery needs real implementation. The demo code returns OTPs in API responses for testing. Production must send OTPs through secure out-of-band channels: SMS via Twilio or AWS SNS, email via SendGrid or AWS SES, or time-based codes via authenticator apps implementing the TOTP standard.

Monitoring and alerting become critical at scale. Forward security events to SIEM systems like Splunk, Elastic Stack, or Datadog. Implement real-time alerting for suspicious patterns: multiple authentication failures, unexpected geographic locations, unusual access times, rapid role switching. Consider machine learning-based anomaly detection for advanced threat detection.

Compliance requirements vary by jurisdiction and industry. HIPAA requires seven-year audit log retention and encrypted data at rest. GDPR requires data masking, right to erasure, and explicit consent tracking. Implement append-only audit tables with cryptographic integrity verification. Create separate database replicas for audit data to prevent tampering.

High availability requires database replication, connection pooling, and load balancing. Configure PostgreSQL streaming replication for read replicas and automatic failover. Use HikariCP or another production-grade connection pool. Deploy multiple application instances behind nginx or HAProxy with health checks and session affinity.

DDoS protection operates at multiple levels. Deploy a web application firewall (AWS WAF, Cloudflare, Akamai) to filter malicious traffic before it reaches applications. Configure rate limiting at the edge to prevent overwhelming backend systems. Implement request size limits and timeouts to prevent resource exhaustion attacks.

Extending the Architecture

This architecture extends naturally to new security requirements through the same patterns. Suppose you need IP whitelisting for administrative functions. Create an interceptor binding annotation:

```
@InterceptorBinding
@Retention(RUNTIME)
@Target({METHOD, TYPE})
public @interface RequireIpWhitelist {
    String[] allowedIps();
}
```

Implement the interceptor:

```
@Interceptor
@RequireIpWhitelist(allowedIps = {})
@Priority(Interceptor.Priority.APPLICATION)
public class IpWhitelistInterceptor {
    @Inject
    private HttpServletRequest request;
    @AroundInvoke
    public Object checkIp(InvocationContext ctx) throws Exception {
        RequireIpWhitelist annotation = ctx.getMethod().
getAnnotation(RequireIpWhitelist.class);
```

```
        String clientIp = request.getRemoteAddr();

        if (!Arrays.asList(annotation.allowedIps()).contains(clientIp)) {

            throw new ForbiddenException("IP not whitelisted");

        }

        return ctx.proceed();

    }

}
```

Register the interceptor in beans.xml and apply it to protected resources. The pattern remains consistent: annotation defines requirements, interceptor enforces them, priority controls execution order.

Custom security events follow a similar pattern. Extend the event type enumeration, fire events using the injected publisher, observe events with @ObservesAsync methods. The loose coupling between event publishers and observers enables independent evolution of security enforcement and security monitoring.

Conclusions

Zero Trust architecture with Jakarta EE and MicroProfile shows that comprehensive security doesn't require sacrificing developer productivity or application maintainability. Using the standard APIs and annotations, your application can achieve zero trust security in a maintainable and extendable way.

The key insight is separation of concerns. Jakarta Security handles authentication and authorization primitives. MicroProfile JWT manages token validation. Bean Validation protects data integrity. Jakarta CDI events enable monitoring. Interceptors extend these standards for domain-specific requirements. Each concern has a dedicated API, preventing security logic from scattering through and getting coupled with business code.

Every endpoint composes multiple independent security layers: authentication proves identity, authorization enforces access control, auditing maintains compliance trails, encryption protects data in transit, validation ensures data integrity. These layers work together yet fail independently, preventing single points of failure.

The patterns shown here apply beyond the healthcare example. Any enterprise application requiring strong security can use these same techniques. The APIs are standardized, the patterns are well-understood, and the implementations are portable across Jakarta EE runtimes like Payara Server.

Production deployment requires hardening around distributed state, secret management, and monitoring infrastructure. The patterns provide the foundation; production systems need Redis for shared state, proper secret management for credentials, real MFA delivery mechanisms, and comprehensive monitoring infrastructure.

Try Payara for Production-Ready Jakarta EE

Ready to deploy your Zero Trust architecture? **Payara Platform Enterprise** is a modern, production-hardened runtime specifically designed for Jakarta EE and MicroProfile applications. Unlike generic application servers, Payara provides enterprise-grade features built for mission-critical deployments:

Why Payara for Zero Trust Applications:

- **Full Jakarta EE & MicroProfile support** - Run all the security patterns in this guide without modification
- **Production-ready out of the box** - Advanced monitoring, clustering, and health checks includedt
- **Security-first design** - Built-in support for OAuth2, OpenID Connect, and certificate management
- **Zero downtime deployments** - Rolling updates and session replication for high availability
- **Enterprise support available** - Get expert help when you need it most

Get Started:

- **Payara Server** - Full-featured application server for traditional deployments
- **Payara Micro** - Lightweight runtime perfect for containers and microservices
- **Payara Qube** - Fully managed Jakarta EE/Spring Boot and Quarkus deployments in the cloud

Download Payara Platform Enteprise today at payara.fish and see why organizations worldwide trust Payara for their mission-critical enterprise Java applications.

info@payara.fish

UK: +44 800 538 5490
Intl: +1 888 239 8941

www.payara.fish