APEX Protocol

Asymmetric Protection with Ephemeral X-keys

Official Technical Documentation

Version 1.0.0 | Spring Boot Edition | TARQEN

End-to-end encryption for every API request between your React frontend and your Spring Boot backend — transparent, lightweight, and production-ready.

GitHub: github.com/TARQEN/apex

Documentation: https://tarqen.com/apex

Section 1 — What is APEX?

APEX (Asymmetric Protection with Ephemeral X-keys) is an open-source end-to-end encryption protocol designed specifically for web applications. It secures every API request and response between a React frontend and a Spring Boot backend using a combination of elliptic curve key exchange, AES-256-GCM authenticated encryption, and sequence-based replay protection — without requiring any changes to your existing controllers, routes, or business logic.

APEX sits invisibly between your frontend fetch calls and your backend endpoints. The frontend sends an encrypted packet. The backend decrypts it, processes it normally, encrypts the response, and sends it back. Your UserController never knows any of this happened.

When a user's browser first connects, APEX performs a one-time handshake where both sides generate temporary key pairs, exchange only their public halves, and independently compute the same shared secret using ECDH mathematics. This secret never travels over the network. Two keys are then derived using HKDF — one for encrypting data, one for authenticating it. Every subsequent API call is encrypted, stamped with a sequence number, and sealed with an authentication tag. The session key is discarded when the session ends.

1.1 Why APEX exists

HTTPS encrypts the transport layer but does not protect against: stolen JWT tokens, compromised load balancers or proxies, man-in-the-middle attacks from internal infrastructure, replay attacks where captured valid requests are re-sent, or request tampering where bytes are modified in transit. APEX eliminates all of these at the protocol level. An attacker who intercepts every packet sees only random bytes — no endpoints, no tokens, no user data, no request structure.

1.2 Security properties

Property	Mechanism	What it stops
Confidentiality	AES-256-GCM encryption	Nobody can read request or response content
Integrity	GCM authentication tag	Any modification to even 1 bit is detected instantly
Authentication	HMAC-SHA256 signing	Requests can only come from sessions with valid keys
Replay protection	Sequence numbers	Recorded packets cannot be re-sent
Forward secrecy	Ephemeral ECDH keys	Past sessions stay safe even after future breach

Section 2 — How APEX works

APEX operates in four sequential phases. Understanding these phases is essential before integration.

2.1 Phase 1 — The handshake (key exchange)

The handshake runs once per session. It establishes the shared encryption keys without ever transmitting them over the network.

1. Frontend generates an ephemeral ECDH keypair (P-256 curve): priv_c and pub_c. It also generates a random 16-byte nonce_c.
2. Frontend sends pub_c and nonce_c to POST /apex/handshake. The private key priv_c never leaves the browser.
3. Backend (ApexHandshakeController.java) generates its own ephemeral keypair: priv_s and pub_s. It generates nonce_s. It sends pub_s, nonce_s, and a sessionId back.
4. Both sides independently compute the same shared secret: Frontend computes ECDH(priv_c, pub_s). Backend computes ECDH(priv_s, pub_c). ECDH mathematics guarantees both computations produce identical bytes.
5. Both sides run HKDF-SHA256 on the shared secret with both nonces as salt. This produces two 256-bit keys: enc_key (for AES-256-GCM encryption) and mac_key (for HMAC-SHA256 signing).
6. The ephemeral private keys are discarded. The session is now active.

> *The attacker watching the wire sees pub_c, pub_s, and both nonces. To compute the shared secret they need either priv_c or priv_s — both of which stayed locked on their respective machines. This is the discrete logarithm problem and is computationally impossible to brute force at P-256 key sizes.*

2.2 Phase 2 — Encrypting a request

Every API call from the frontend is wrapped in an APEX envelope before transmission.

Step	Serialize the request payload to JSON bytes
AES-256-GCM	Encrypt bytes using enc_key and a fresh random 12-byte IV
Auth tag	AES-GCM automatically produces a 16-byte authentication tag
Sequence	Include the current seq number (starts at 1, increments each request)
Wire format	Send { iv, ciphertext, authTag, seq } as the request body

2.3 Phase 3 — Server verification and decryption

The ApexFilter.java intercepts every incoming request and runs four checks in order. If any check fails, the request is rejected with HTTP 401 immediately.

7. Session check: reads X-Apex-Session header, looks up session in ApexSessionStore. If session not found or expired, reject.
8. Auth tag check: AES-GCM decryption verifies the authentication tag automatically. If the tag does not match, even one bit was tampered with — throws AEADBadTagException, reject.

9. Sequence number check: compares the decrypted seq value against the expected next seq stored in the session. If they do not match, this is a replay attack, reject.
10. Decryption: all checks passed — plaintext bytes are recovered and wrapped in a new request object. Your controller receives this exactly as if APEX never existed.

2.4 Phase 4 — Encrypting the response

After your controller returns a response, the filter intercepts it on the way out. The response body is encrypted with enc_key and a fresh IV, wrapped in the same APEX envelope format, and sent back. The frontend decrypts it using the same enc_key derived during the handshake.

2.5 Key rotation

Sessions automatically expire after 15 minutes or 1000 requests, whichever comes first. The client must re-perform the handshake to get a new session. This limits the window of exposure if a session is somehow compromised. The ApexSessionStore runs a scheduled cleanup every 5 minutes to evict expired sessions from memory.

Section 3 — Prerequisites

3.1 Backend requirements

Java version	Java 17 or higher
Spring Boot	3.x (Jakarta EE namespace)
Build tool	Maven or Gradle
New dependency	BouncyCastle bcprov-jdk18on 1.78.1 (for HKDF)
Lombok	Already in your project (required)
Scheduling	@EnableScheduling for session cleanup (no extra dependency)

3.2 Frontend requirements

Browser API	window.crypto.subtle (available in all modern browsers)
Framework	React 17+ (or any framework — vanilla JS also works)
New npm packages	None — zero additional dependencies
Node.js (dev)	Any version for your build tool

3.3 Infrastructure

Single instance	ConcurrentHashMap in memory — no additional setup
Multiple instances	Replace session store with Redis (ElastiCache on AWS)
AWS EBS	No changes to Docker or deployment process
HTTPS	Required in production (APEX adds a second layer on top)

> *APEX is designed to be used on top of HTTPS, not instead of it. In production, always deploy behind TLS. APEX provides payload-level encryption; HTTPS provides transport-level encryption. Both together give you defense in depth.*

Section 4 — Backend integration (Spring Boot)

4.1 Maven dependency

Add this single dependency to your pom.xml. Everything else uses Java's built-in javax.crypto package.

```
<dependency>
    <groupId>org.bouncycastle</groupId>
    <artifactId>bcprov-jdk18on</artifactId>
    <version>1.78.1</version>
</dependency>
```

4.2 File structure

Create the following files inside your project. Replace com.yourapp with your actual package name.

```
src/main/java/com/yourapp/
  apex/
    ApexSession.java           <- Lombok model: session data
    ApexSessionStore.java      <- ConcurrentHashMap session store
    ApexCryptoService.java     <- All crypto: ECDH, HKDF, AES-GCM
    ApexHandshakeController.java  <- POST /apex/handshake endpoint
    ApexFilter.java            <- Intercepts every API request
    ApexConfig.java            <- Registers filter + enables scheduling
  dto/
    HandshakeRequest.java      <- Lombok DTO
    HandshakeResponse.java     <- Lombok DTO
    ApexEnvelope.java          <- Wire packet format
```

4.3 ApexSession.java

> *This class holds everything the server needs per active session. @Builder.Default ensures fields get their default values even when using the builder pattern.*

```
package com.yourapp.apex;

import lombok.Builder;
import lombok.Data;
import javax.crypto.SecretKey;
import java.time.Instant;

@Data
@Builder
public class ApexSession {
    private SecretKey encKey;
    private SecretKey macKey;
    private String sessionId;
    private Instant createdAt;

    @Builder.Default
    private volatile long expectedSeq = 1L;
```

```
    @Builder.Default
    private int requestCount = 0;

    public synchronized void incrementSeq() {
        this.expectedSeq++;
        this.requestCount++;
    }

    public boolean isExpired() {
        boolean tooOld = Instant.now()
            .isAfter(createdAt.plusSeconds(900));
        boolean tooMany = requestCount >= 1000;
        return tooOld || tooMany;
    }
}
```

4.4 ApexSessionStore.java

> *Uses ConcurrentHashMap — no Redis required. The @Scheduled method automatically evicts expired sessions every 5 minutes. For multi-instance deployments on AWS, replace the ConcurrentHashMap with a Redis template pointing to ElastiCache.*

```
package com.yourapp.apex;

import lombok.extern.slf4j.Slf4j;
import org.springframework.scheduling.annotation.Scheduled;
import org.springframework.stereotype.Component;
import java.util.concurrent.ConcurrentHashMap;

@Slf4j
@Component
public class ApexSessionStore {

    private final ConcurrentHashMap<String, ApexSession> sessions
        = new ConcurrentHashMap<>();

    public void save(String id, ApexSession s) { sessions.put(id, s); }
    public ApexSession get(String id)          { return sessions.get(id); }
    public void remove(String id)              { sessions.remove(id); }
    public boolean exists(String id)           { return sessions.containsKey(id); }

    @Scheduled(fixedDelay = 300_000)
    public void evictExpired() {
        int before = sessions.size();
        sessions.entrySet().removeIf(e -> e.getValue().isExpired());
        int removed = before - sessions.size();
        if (removed > 0)
            log.info("Evicted {} expired APEX sessions", removed);
```

```
    }
}
```

4.5 ApexCryptoService.java

> *All cryptography lives here. No HTTP, no Spring annotations — pure maths. This class is the mirror of apexCrypto.js on the frontend.*

```
package com.yourapp.apex;

import org.bouncycastle.crypto.generators.HKDFBytesGenerator;
import org.bouncycastle.crypto.params.HKDFParameters;
import org.bouncycastle.crypto.digests.SHA256Digest;
import org.springframework.stereotype.Service;

import javax.crypto.*;
import javax.crypto.spec.*;
import java.nio.ByteBuffer;
import java.security.*;
import java.security.spec.*;
import java.util.Arrays;
import java.util.Base64;

@Service
public class ApexCryptoService {

    private static final int GCM_TAG_BITS = 128;
    private static final int GCM_IV_BYTES  = 12;
    private static final int KEY_BYTES     = 32;

    public KeyPair generateECDHKeyPair() throws Exception {
        KeyPairGenerator kpg = KeyPairGenerator.getInstance("EC");
        kpg.initialize(new ECGenParameterSpec("secp256r1"));
        return kpg.generateKeyPair();
    }

    public byte[] computeSharedSecret(
            PrivateKey myPriv, PublicKey theirPub) throws Exception {
        KeyAgreement ka = KeyAgreement.getInstance("ECDH");
        ka.init(myPriv);
        ka.doPhase(theirPub, true);
        return ka.generateSecret();
    }

    public SecretKey deriveKey(byte[] sharedSecret,
            byte[] nonceC, byte[] nonceS, String usage) {
        byte[] salt = concat(nonceC, nonceS);
        byte[] info = usage.getBytes();
        HKDFBytesGenerator hkdf =
```

```
            new HKDFBytesGenerator(new SHA256Digest());
        hkdf.init(new HKDFParameters(sharedSecret, salt, info));
        byte[] key = new byte[KEY_BYTES];
        hkdf.generateBytes(key, 0, KEY_BYTES);
        return new SecretKeySpec(key, "AES");
    }

    public byte[] encrypt(SecretKey key, byte[] iv,
            byte[] plaintext) throws Exception {
        Cipher c = Cipher.getInstance("AES/GCM/NoPadding");
        c.init(Cipher.ENCRYPT_MODE, key,
            new GCMParameterSpec(GCM_TAG_BITS, iv));
        return c.doFinal(plaintext);
    }

    public byte[] decrypt(SecretKey key, byte[] iv,
            byte[] ciphertext) throws Exception {
        Cipher c = Cipher.getInstance("AES/GCM/NoPadding");
        c.init(Cipher.DECRYPT_MODE, key,
            new GCMParameterSpec(GCM_TAG_BITS, iv));
        return c.doFinal(ciphertext);
    }

    public byte[] generateIV() {
        byte[] iv = new byte[GCM_IV_BYTES];
        new SecureRandom().nextBytes(iv);
        return iv;
    }

    public String encodePublicKey(PublicKey key) {
        return Base64.getEncoder()
            .encodeToString(key.getEncoded());
    }

    public PublicKey decodePublicKey(String base64)
            throws Exception {
        byte[] bytes = Base64.getDecoder().decode(base64);
        return KeyFactory.getInstance("EC")
            .generatePublic(new X509EncodedKeySpec(bytes));
    }

    public byte[] randomBytes(int n) {
        byte[] b = new byte[n];
        new SecureRandom().nextBytes(b);
        return b;
    }

    private byte[] concat(byte[]... arrays) {
```

```
            int total = Arrays.stream(arrays)
                .mapToInt(a -> a.length).sum();
            ByteBuffer buf = ByteBuffer.allocate(total);
            for (byte[] a : arrays) buf.put(a);
            return buf.array();
        }
    }
```

4.6 DTOs

```
// HandshakeRequest.java
@Data
public class HandshakeRequest {
    private String pubC;    // frontend public key (Base64)
    private String nonceC;  // frontend nonce (Base64)
}

// HandshakeResponse.java
@Data @Builder
public class HandshakeResponse {
    private String sessionId;
    private String pubS;    // server public key (Base64)
    private String nonceS;  // server nonce (Base64)
}

// ApexEnvelope.java  (the wire format for every API call)
@Data
public class ApexEnvelope {
    private String iv;          // 12-byte IV (Base64)
    private String ciphertext;  // AES-GCM encrypted body (Base64)
    private long   seq;         // sequence number
}
```

4.7 ApexHandshakeController.java

```
package com.yourapp.apex;

import com.yourapp.dto.*;
import lombok.RequiredArgsConstructor;
import lombok.extern.slf4j.Slf4j;
import org.springframework.http.ResponseEntity;
import org.springframework.web.bind.annotation.*;

import javax.crypto.SecretKey;
import java.security.*;
import java.time.Instant;
import java.util.Base64;
import java.util.UUID;
```

```
@Slf4j
@RestController
@RequestMapping("/apex")
@RequiredArgsConstructor
public class ApexHandshakeController {

    private final ApexCryptoService  crypto;
    private final ApexSessionStore   store;

    @PostMapping("/handshake")
    public ResponseEntity<HandshakeResponse> handshake(
            @RequestBody HandshakeRequest req) {
        try {
            PublicKey clientPub =
                crypto.decodePublicKey(req.getPubC());
            byte[] nonceC =
                Base64.getDecoder().decode(req.getNonceC());

            KeyPair serverKeys = crypto.generateECDHKeyPair();
            byte[] nonceS = crypto.randomBytes(16);

            byte[] shared = crypto.computeSharedSecret(
                serverKeys.getPrivate(), clientPub);

            SecretKey encKey =
                crypto.deriveKey(shared, nonceC, nonceS, "enc");
            SecretKey macKey =
                crypto.deriveKey(shared, nonceC, nonceS, "mac");

            String sessionId = UUID.randomUUID().toString();
            ApexSession session = ApexSession.builder()
                .sessionId(sessionId)
                .encKey(encKey)
                .macKey(macKey)
                .createdAt(Instant.now())
                .build();
            store.save(sessionId, session);

            return ResponseEntity.ok(
                HandshakeResponse.builder()
                    .sessionId(sessionId)
                    .pubS(crypto.encodePublicKey(
                        serverKeys.getPublic()))
                    .nonceS(Base64.getEncoder()
                        .encodeToString(nonceS))
                    .build());
        } catch (Exception e) {
            log.error("Handshake failed", e);
```

```
                return ResponseEntity.badRequest().build();
            }
        }
    }
```

4.8 ApexFilter.java

> *This filter intercepts every /api/* request. It runs 4 checks, decrypts the body, passes the plain request to your controller, then encrypts the response on the way back out. Your controllers require zero changes.*

```
package com.yourapp.apex;

import com.fasterxml.jackson.databind.ObjectMapper;
import com.yourapp.dto.ApexEnvelope;
import jakarta.servlet.*;
import jakarta.servlet.http.*;
import lombok.RequiredArgsConstructor;
import lombok.extern.slf4j.Slf4j;
import org.springframework.stereotype.Component;

import java.io.*;
import java.util.Base64;
import java.util.Set;

@Slf4j
@Component
@RequiredArgsConstructor
public class ApexFilter implements Filter {

    private final ApexCryptoService crypto;
    private final ApexSessionStore  store;
    private final ObjectMapper      mapper;

    private static final Set<String> SKIP = Set.of(
        "/apex/handshake", "/actuator/health", "/actuator/info");

    @Override
    public void doFilter(ServletRequest req,
            ServletResponse res, FilterChain chain)
            throws IOException, ServletException {

        HttpServletRequest  hReq = (HttpServletRequest)  req;
        HttpServletResponse hRes = (HttpServletResponse) res;

        if (SKIP.contains(hReq.getRequestURI())) {
            chain.doFilter(req, res);
            return;
        }
```

```
try {
    // Check 1: session
    String sid = hReq.getHeader("X-Apex-Session");
    if (sid == null) { reject(hRes, "No session"); return; }
    ApexSession s = store.get(sid);
    if (s == null)   { reject(hRes, "Session not found"); return; }
    if (s.isExpired()) {
        store.remove(sid);
        reject(hRes, "Session expired"); return;
    }

    // Parse envelope
    ApexEnvelope env = mapper.readValue(
        hReq.getInputStream().readAllBytes(),
        ApexEnvelope.class);
    byte[] iv = Base64.getDecoder().decode(env.getIv());
    byte[] ct = Base64.getDecoder()
        .decode(env.getCiphertext());

    synchronized (s) {
        // Check 2+3: auth tag + seq (decrypt verifies tag)
        if (env.getSeq() != s.getExpectedSeq()) {
            reject(hRes, "Replay detected"); return;
        }
        byte[] plain = crypto.decrypt(s.getEncKey(), iv, ct);
        s.incrementSeq();

        // Pass decrypted request to controller
        BodyReplacedRequest wrapped =
            new BodyReplacedRequest(hReq, plain);
        BodyCapturingResponse capture =
            new BodyCapturingResponse(hRes);
        chain.doFilter(wrapped, capture);

        // Encrypt response
        byte[] respBytes = capture.getCapturedBody();
        byte[] rIv = crypto.generateIV();
        byte[] rCt = crypto.encrypt(
            s.getEncKey(), rIv, respBytes);

        ApexEnvelope out = new ApexEnvelope();
        out.setIv(Base64.getEncoder().encodeToString(rIv));
        out.setCiphertext(Base64.getEncoder()
            .encodeToString(rCt));
        out.setSeq(s.getExpectedSeq());

        hRes.setContentType("application/json");
        hRes.getWriter().write(
```

```
                    mapper.writeValueAsString(out));
            }
        } catch (javax.crypto.AEADBadTagException e) {
            log.warn("APEX tamper detected");
            reject(hRes, "Authentication failed");
        } catch (Exception e) {
            log.error("APEX filter error", e);
            reject(hRes, "APEX error");
        }
    }

    private void reject(HttpServletResponse r, String msg)
            throws IOException {
        r.setStatus(401);
        r.setContentType("application/json");
        r.getWriter().write("{\"error\":\"" + msg + "\"}");
    }

    // Inner classes: BodyReplacedRequest and
    // BodyCapturingResponse omitted for brevity.
    // Full implementations in GitHub repository.
}
```

4.9 ApexConfig.java

```
package com.yourapp.apex;

import org.springframework.boot.web.servlet.FilterRegistrationBean;
import org.springframework.context.annotation.*;
import org.springframework.scheduling.annotation.EnableScheduling;

@Configuration
@EnableScheduling
public class ApexConfig {

    @Bean
    public FilterRegistrationBean<ApexFilter> apexFilter(
            ApexFilter filter) {
        FilterRegistrationBean<ApexFilter> reg =
            new FilterRegistrationBean<>();
        reg.setFilter(filter);
        reg.addUrlPatterns("/api/*");
        reg.setOrder(1);
        return reg;
    }
}
```

Section 5 — Frontend integration (React)

The frontend requires zero npm packages. Everything uses the browser's built-in window.crypto.subtle API.

5.1 File structure

```
src/
  apex/
    apexCrypto.js    <- raw crypto: ECDH, HKDF, AES-GCM
    apexSession.js   <- in-memory session store
    apexClient.js    <- main wrapper: handshake + fetch
  api/
    userApi.js       <- your existing API calls (minimal change)
  components/
    UserProfile.jsx  <- your components (zero change)
```

5.2 apexCrypto.js

> *Every function here mirrors the equivalent method in ApexCryptoService.java. The curve (P-256 = secp256r1), hash (SHA-256), and key sizes are identical on both sides.*

```
const subtle = window.crypto.subtle;

export async function generateECDHKeyPair() {
  return subtle.generateKey(
    { name: "ECDH", namedCurve: "P-256" },
    true, ["deriveKey"]
  );
}

export async function exportPublicKey(publicKey) {
  const raw = await subtle.exportKey("spki", publicKey);
  return arrayBufferToBase64(raw);
}

export async function importPublicKey(base64) {
  return subtle.importKey("spki", base64ToArrayBuffer(base64),
    { name: "ECDH", namedCurve: "P-256" }, true, []);
}

export async function computeSharedSecret(myPriv, theirPub) {
  return subtle.deriveKey(
    { name: "ECDH", public: theirPub }, myPriv,
    { name: "AES-GCM", length: 256 }, true,
    ["encrypt", "decrypt"]
  );
}

export async function hkdfDeriveKey(
```

```
    sharedKey, nonceC, nonceS, usage) {
  const raw = await subtle.exportKey("raw", sharedKey);
  const km  = await subtle.importKey("raw", raw,
    { name: "HKDF" }, false, ["deriveKey"]);
  const salt = concatBuffers(nonceC, nonceS);
  const info = new TextEncoder().encode(usage);
  return subtle.deriveKey(
    { name: "HKDF", hash: "SHA-256", salt, info },
    km, { name: "AES-GCM", length: 256 }, false,
    ["encrypt", "decrypt"]
  );
}

export async function aesGcmEncrypt(encKey, plaintextBytes) {
  const iv = crypto.getRandomValues(new Uint8Array(12));
  const ct = await subtle.encrypt(
    { name: "AES-GCM", iv }, encKey, plaintextBytes);
  return { iv: arrayBufferToBase64(iv),
           ciphertext: arrayBufferToBase64(ct) };
}

export async function aesGcmDecrypt(encKey, ivB64, ctB64) {
  const pt = await subtle.decrypt(
    { name: "AES-GCM", iv: base64ToArrayBuffer(ivB64) },
    encKey, base64ToArrayBuffer(ctB64));
  return new Uint8Array(pt);
}

export function generateNonce() {
  const n = crypto.getRandomValues(new Uint8Array(16));
  return { bytes: n, base64: arrayBufferToBase64(n) };
}
```

5.3 apexClient.js

> *This is the only file your application code interacts with. It auto-performs the handshake on first call and retries once if the session expires. Your API layer calls apex.get() and apex.post() exactly like fetch().*

```
import * as crypto  from "./apexCrypto.js";
import * as session from "./apexSession.js";

const BASE_URL = import.meta.env.VITE_API_URL
    || "http://localhost:8080";

async function performHandshake() {
  const keyPair = await crypto.generateECDHKeyPair();
  const pubC    = await crypto.exportPublicKey(keyPair.publicKey);
  const { bytes: ncBytes, base64: ncB64 } =
      crypto.generateNonce();
```

```
  const res = await fetch(`${BASE_URL}/apex/handshake`, {
    method: "POST",
    headers: { "Content-Type": "application/json" },
    body: JSON.stringify({ pubC, nonceC: ncB64 })
  });
  if (!res.ok) throw new Error("Handshake failed");

  const { sessionId, pubS, nonceS } = await res.json();
  const serverPub = await crypto.importPublicKey(pubS);
  const shared    = await crypto.computeSharedSecret(
      keyPair.privateKey, serverPub);
  const nsBytes = Uint8Array.from(atob(nonceS),
      c => c.charCodeAt(0));

  const encKey = await crypto.hkdfDeriveKey(
      shared, ncBytes, nsBytes, "enc");
  const macKey = await crypto.hkdfDeriveKey(
      shared, ncBytes, nsBytes, "mac");

  session.saveSession({ sessionId, encKey, macKey });
}

export async function apexFetch(path, options = {}) {
  if (!session.hasSession()) await performHandshake();

  const s = session.getSession();
  const plain = new TextEncoder().encode(JSON.stringify({
    method:  options.method  || "GET",
    path,
    headers: options.headers || {},
    body:    options.body    || null
  }));

  const seq = session.getAndIncrementSeq();
  const { iv, ciphertext } =
      await crypto.aesGcmEncrypt(s.encKey, plain);

  const resp = await fetch(`${BASE_URL}/api/secure`, {
    method: "POST",
    headers: {
      "Content-Type":   "application/json",
      "X-Apex-Session": s.sessionId
    },
    body: JSON.stringify({ iv, ciphertext, seq })
  });
```

```
  if (resp.status === 401) {
    session.clearSession();
    return apexFetch(path, options);
  }

  const env = await resp.json();
  const dec = await crypto.aesGcmDecrypt(
      s.encKey, env.iv, env.ciphertext);
  return JSON.parse(new TextDecoder().decode(dec));
}

export const apex = {
  get:    path       => apexFetch(path, { method: "GET" }),
  post:   (path, d)  => apexFetch(path, { method: "POST",
      body: JSON.stringify(d) }),
  put:    (path, d)  => apexFetch(path, { method: "PUT",
      body: JSON.stringify(d) }),
  delete: path       => apexFetch(path, { method: "DELETE" })
};
```

5.4 Using apex in your components

Before APEX, your API calls looked like this:

```
const res  = await fetch("/api/user/profile");
const data = await res.json();
```

After APEX, they look like this:

```
import { apex } from "../apex/apexClient.js";

const data = await apex.get("/api/user/profile");
// Handshake, encryption, decryption all automatic.
// data is the plain JSON object your controller returned.
```

Your React components require zero changes. Only the API layer changes, and only by one import and one function call.

Section 6 — Security reference

6.1 Cryptographic primitives

Primitive	Algorithm	Parameters	Purpose
Key exchange	ECDH	P-256 (secp256r1)	Derive shared secret without transmitting it
Key derivation	HKDF-SHA256	256-bit output, 2 keys	Derive enc_key and mac_key from shared secret
Encryption	AES-256-GCM	256-bit key, 96-bit IV	Encrypt + authenticate every request/response
IV generation	CSPRNG	12 bytes per request	Ensure unique encryption per request
Session nonces	CSPRNG	16 bytes each side	Ensure unique key derivation per session

6.2 Attack resistance

Attack	How APEX blocks it
Eavesdropping	AES-256-GCM — all content is ciphertext. No endpoints, tokens, or data are readable.
Man in the middle	ECDH shared secret requires both private keys. An interceptor cannot compute it.
Token theft	Tokens are inside the encrypted payload. They never appear in plaintext on the wire.
Replay attack	Sequence numbers are encrypted inside the payload. Old seq values are permanently rejected.
Tampering	GCM authentication tag covers the entire ciphertext. 1-bit change causes immediate rejection.
Session hijacking	Session IDs alone are worthless — requests still require valid enc_key to produce a valid auth tag.
Brute force keys	P-256 provides 128-bit security. More possible keys than atoms in the observable universe.
Future breach	Forward secrecy — ephemeral keys are discarded after each session. Past traffic cannot be decrypted.

6.3 Known limitations

- APEX does not replace TLS. Always use HTTPS in production. APEX is a second layer on top.
- Single-instance only by default. For auto-scaled deployments with multiple instances, replace ConcurrentHashMap with Redis so all instances share session state.
- APEX does not provide user authentication. It provides request confidentiality. You still need JWT or session-based auth for identifying users.

- The frontend session lives in browser memory. It is cleared when the tab closes. Mobile apps may need a different session persistence strategy.

Section 7 — Deployment on AWS EBS

APEX requires no changes to your Docker image, Dockerfile, or EBS deployment process. It is just Java classes inside your existing jar.

7.1 Single instance deployment

No additional configuration required. Deploy exactly as you normally would. The ConcurrentHashMap session store works perfectly for single-instance deployments.

7.2 Multi-instance (auto-scaled) deployment

When EBS auto-scales to multiple instances, sessions created on instance A will not exist on instance B. Solve this by replacing the ConcurrentHashMap with Redis:

```
// In application.properties
spring.data.redis.host=${REDIS_HOST}
spring.data.redis.port=6379

// Replace ApexSessionStore with Redis-backed version
// Full implementation in GitHub repository under
// apex-spring-boot/src/main/java/.../ApexRedisSessionStore.java
```

Set the environment variable in your EBS configuration:

```
REDIS_HOST = your-elasticache-endpoint.cache.amazonaws.com
```

7.3 Environment variables

VITE_API_URL	Your Spring Boot backend URL (frontend .env)
SERVER_PORT	Spring Boot port (default 8080)

Section 8 — Troubleshooting

Error	Cause	Fix
401 Session not found	Session expired or never created	Re-run handshake. Check session TTL settings.
401 Replay detected	Seq number out of order	Do not reuse ApexEnvelope objects. Each request needs a fresh seq.
401 Authentication failed	GCM auth tag mismatch	Packet was tampered or enc_key mismatch. Check HKDF inputs match on both sides.
AEADBadTagException	Wrong key or modified ciphertext	Verify nonceC and nonceS are passed in the same order to HKDF on both sides.
Handshake 400	Invalid public key format	Ensure pub_c is exported as SPKI Base64 on the frontend.
CORS error on /apex/handshake	Missing CORS config	Add /apex/handshake to your Spring Boot CORS allowed paths.
Session lost on tab refresh	In-memory session store	Expected behaviour. The frontend re-handshakes automatically on next request.

Section 9 — Comparison with similar protocols

Feature	APEX	HTTPS only	MTProto (Telegram)	TLS 1.3
Payload encryption	Yes	No	Yes	No
Key exchange	ECDH P-256	N/A	RSA + DH	ECDH
Encryption	AES-256-GCM	N/A	AES-256-IGE	AES-GCM
Forward secrecy	Yes	No	Yes	Yes
Replay protection	Seq numbers	No	msgid + salt	No
Auth tag	GCM built-in	N/A	SHA-1 MAC	GCM built-in
Spring Boot integration	Native (1 filter)	N/A	No	Via TLS config
Frontend dependency	Zero (crypto.subtle)	N/A	Heavy library	N/A
Open source	Yes (MIT)	N/A	Yes (GPL)	Standard

Section 10 — GitHub repository and resources

10.1 Repository structure

```
github.com/apex-protocol/apex

apex/
  apex-spring-boot/        <- Spring Boot starter library
    src/main/java/         <- All Java source files
    src/test/java/         <- Unit and integration tests
    pom.xml

  apex-react/              <- React client library
    src/apex/              <- apexCrypto.js, apexClient.js
    src/api/               <- Example API layer
    package.json

  demo/
    spring-boot-demo/      <- Full working Spring Boot demo app
    react-demo/            <- Full working React demo app
    docker-compose.yml     <- Run the demo with one command

  docs/
    APEX-v1.0.0-Documentation.docx
    README.md
    SECURITY.md
    CONTRIBUTING.md
```

10.2 Quick start with the demo

11. Clone the repository: git clone https://github.com/Tarqen/APEX.git
12. Start the demo: cd apex/demo && docker-compose up
13. Open http://localhost:5173 in your browser
14. Open browser DevTools Network tab and watch all requests
15. Every API call shows as encrypted ciphertext in the network tab

10.4 Reporting security issues

Do not open a public GitHub issue for security vulnerabilities. Email support@tarqen.com with a description of the vulnerability. We will respond within 48 hours and credit you in the release notes.

APEX Protocol v1.0.0 — Tarqen Solutions — Open Source

Built with care for every developer who deserves genuinely private API traffic.

github.com/TARQEN/apex | docs.apex-protocol.com | support@tarqen.com