National College of Ireland

Project Submission Sheet

Student Name:	Eskandar Atrakchi
Student ID:	X23137517
Programme:	**Secure Application Programming (BSHCSD4)** **Year:** 2025
Module:	**BSHCSD4**
Lecturer:	Eugene McLaughlin
Submission Due Date:	31st/Oct/2025
Project Title:	Java Authentication and Session Management System
Word Count:	……

I hereby certify that the information contained in this (my submission) is information pertaining to research I conducted for this project. All information other than my own contribution will be fully referenced and listed in the relevant bibliography section at the rear of the project.
ALL internet material must be referenced in the references section. Students are encouraged to use the Harvard Referencing Standard supplied by the Library. To use other author's written or electronic work is illegal (plagiarism) and may result in disciplinary action. Students may be required to undergo a viva (oral examination) if there is suspicion about the validity of their submitted work.

Signature:	Eskandar Atrakchi ……
Date:	27th/Oct/2025 ……

PLEASE READ THE FOLLOWING INSTRUCTIONS:

1. Please attach a completed copy of this sheet to each project (including multiple copies).
2. Projects should be submitted to your Programme Coordinator.
3. **You must ensure that you retain a HARD COPY of ALL projects**, both for your own reference and in case a project is lost or mislaid. It is not sufficient to keep a copy on computer. Please do not bind projects or place in covers unless specifically requested.
4. You must ensure that all projects are submitted to your Programme Coordinator on or before the required submission date. **Late submissions will incur penalties.**
5. All projects must be submitted and passed in order to successfully complete the year. **Any project/assignment not submitted will be marked as a fail.**
6. **Please check that you read AI and Academic Integrity Acknowledgement Supplments in this document**

Office Use Only	
Signature:	
Date:	
Penalty Applied (if applicable):	

AI Acknowledgement Supplement

[Insert Module Name]

[Insert Title of your assignment]

Your Name/Student Number	Course	Date
Eskandar Atrakchi	BSHCSD4	31st/Oct/2025

This section is a supplement to the main assignment, to be used if AI was used in any capacity in the creation of your assignment; if you have queries about how to do this, please contact your lecturer. For an example of how to fill these sections out, please click here.

AI Acknowledgment

This section acknowledges the AI tools that were utilized in the process of completing this assignment.

Tool Name	Brief Description	Link to tool
ChatGPT	I asked ChatGPT what hashMap should I use to ensure safe, correct, and efficient access to shared data when multiple threads read and write at the same time, and it recommended me to use ConcurrentHashMap<>();	
ChatGPT	How can I prevent session hijacking and implement token rotation?	

Description of AI Usage

This section provides a more detailed description of how the AI tools were used in the assignment. It includes information about the prompts given to the AI tool, the responses received, and how these responses were utilized or modified in the assignment. **One table should be used for each tool used**.

[Insert Tool Name]	
I used it to learn about token iterations	
How can I prevent session hijacking and implement token rotation?	Use short-lived access tokens + long-lived refresh tokens, rotate (one-time) refresh tokens on every use, store only hashed tokens server-side, bind tokens to context (cookie flags, optional fingerprint), and revoke/blacklist reused tokens — all over HTTPS. Below is a compact, actionable plan plus minimal Java-style snippets you can drop into your code. **1) High-level design (recommended)** • **Access token**: short lifespan (e.g. 5–15 minutes). Returned to client and used for API calls. Can be a random opaque token (what you already do) or a JWT — opaque is fine for server-side state. • **Refresh token**: long lifespan (e.g. days). Stored **only** in an HttpOnly, Secure cookie (or securely on client). Used only to obtain new access tokens.

- **Rotation**: each time a refresh token is used, issue a *new* refresh token and immediately **invalidate the old** one (one-time-use). If an old refresh token is used again (reuse detection), treat as compromise and revoke all sessions for that user.
- **Store only token hashes**: server keeps hash(S) of token, never raw token.
- **Bind session**: store metadata (user-agent, IP range, device id) and compare on refresh/usage; be careful with overly strict IP checks on mobile.
- **Revoke/Blacklist**: keep a revocation store of token hashes to reject old/compromised tokens.
- **Transport / cookies**: always use HTTPS + Secure; HttpOnly; SameSite=Strict (or Lax for cross-site needs).
- **CSRF**: if refresh token is in cookie, use SameSite and/or double-submit CSRF token for refresh endpoint.

---

2) Concrete behaviors to implement

1. **Make access tokens short-lived** (5–15m). Keep them stateless or stored hashed (you already store hashed session tokens).
2. **Issue a refresh token** when user logs in. Store only sha256(refreshToken) with expiry, device info, and a rotation family id.
3. **On refresh request**:
    - Accept raw refresh token from client.
    - Compute hash; look up stored entry.
    - If not found → reject.
    - If found and valid → **generate new access token** and **a brand new refresh token**, store hash(new refresh) and delete the old refresh hash.
    - Return access token and set new refresh token to cookie.
4. **Detect reuse**: if token hash is not found but the token‘s family id exists and indicates previous rotation, treat as reuse (compromise) and revoke all sessions in that family.
5. **Logout / revoke**: delete the stored hash for refresh tokens and access tokens; optionally maintain a short blacklist for recently-seen access token hashes until they expire.

---

3) Minimal Java snippets (fits in your design)

Add a Refresh store entry (hashed token + metadata):

```
private static class RefreshEntry {
  final String username;
  final String familyId;      // groups rotating tokens for same login/device
  final long expiry;
```

```
    // optionally: deviceId, userAgentHash, ipHash
    RefreshEntry(String u, String familyId, long expiry) {
        this.username = u;
        this.familyId = familyId;
        this.expiry = expiry;
    }
}

// store sha256(refreshToken) -> RefreshEntry
private final Map<String, RefreshEntry> refreshStore = new ConcurrentHashMap<>();
```

Issue tokens when logging in:

```
private static final long ACCESS_EXP_MS = 10 * 60 * 1000L; // 10m
private static final long REFRESH_EXP_MS = 7L * 24 * 60 * 60 * 1000L; // 7d

private TokenPair issueTokens(String username) {
    String access = generateSessionToken(); // short lived token (current generateSessionToken)
    String refresh = generateSessionToken(); // longer lived random token
    String refreshHash = sha256Base64(refresh);
    String familyId = sha256Base64(generateSessionToken()); // random family id

    long now = System.currentTimeMillis();
    // store access as before (hashed)
    sessions.put(sha256Base64(access), new Session(username, now + ACCESS_EXP_MS));
    // store refresh hashed with family id
    refreshStore.put(refreshHash, new RefreshEntry(username, familyId, now + REFRESH_EXP_MS));

    return new TokenPair(access, refresh);
}
```

Refresh endpoint (rotation + reuse detection):

```
public TokenPair refresh(String rawRefreshToken, String userAgent) {
    if (rawRefreshToken == null) return null;
    String hash = sha256Base64(rawRefreshToken);
    RefreshEntry entry = refreshStore.get(hash);
    long now = System.currentTimeMillis();

    if (entry == null) {
        // possible reuse: attacker used a stolen refresh token that was already rotated.
        // Optionally: check family id lookup (if you store mapping of families) and revoke all sessions in that family.
        // For simplicity, treat as compromise and return null
```

(and consider alerting/admin revoke).
    return null;
  }

  if (entry.expiry < now) {
    refreshStore.remove(hash);
    return null;
  }

  // VALID: rotate: remove old, create new refresh and new access
  refreshStore.remove(hash);
  String newAccess = generateSessionToken();
  String newRefresh = generateSessionToken();
  String newHash = sha256Base64(newRefresh);
  // keep same family id so reuse detection can trace (or create new family per device)
  refreshStore.put(newHash, new RefreshEntry(entry.username, entry.familyId, now + REFRESH_EXP_MS));

  // store new access token as before
  sessions.put(sha256Base64(newAccess), new Session(entry.username, now + ACCESS_EXP_MS));

  return new TokenPair(newAccess, newRefresh);
}

Detect reuse (advanced): when a refresh hash is used that is not present, but the attacker included a valid token from the same family: maintain another map familyId -> active refresh hashes or last-seen timestamp; if you detect use of a rotated token, immediately revoke family.

4) Cookie & transport recommendations

- Send **refresh token** in cookie:
- Set-Cookie: refresh=<token>; HttpOnly; Secure; SameSite=Strict; Path=/auth/refresh; Max-Age=...
- Send **access token** in Authorization: Bearer <access> header or as JS-readable cookie if you must (less ideal). For SPA, store access token in memory only (not LocalStorage).
- Always use HTTPS.

5) Extra protections

- **Token binding**: tie refresh tokens to a device fingerprint (hash of user-agent + device id) to make reuse harder.
- **Rate-limit** refresh and login endpoints.
- **Monitor reuse**: log refresh token reuse and suspend accounts.
- **Short access token lifetime** reduces window even if stolen.

	• **Rotate on sensitive events**: password change, logout other devices, etc.
	6) Summary checklist (implement these) • Add refresh token store (hashed) + metadata (family/device). • Rotate refresh token on every use (issue new and delete old). • Detect reuse and revoke session family on suspicious reuse. • Use Secure; HttpOnly; SameSite cookies for refresh tokens. • Keep access tokens short-lived and hashed on server. • Use HTTPS and rate-limiting.

Evidence of AI Usage

This section includes evidence of significant prompts and responses used or generated through the AI tool. It should provide a clear understanding of the extent to which the AI tool was used in the assignment. Evidence may be attached via screenshots or text.

Additional Evidence:

How do we use crypto helpers to help developers handle sensitive data securely and consistently without having to repeat the same cryptographic logic everywhere?

Why we use a crypto helper

1. Security Best Practices

o Passwords, tokens, and other sensitive data require careful handling.

o Crypto helpers ensure that secure algorithms (PBKDF2, SHA-256, SecureRandom) are used properly.

2. Reduce Errors

o Cryptography is tricky and easy to get wrong.

o A helper avoids mistakes like reusing salts, weak random numbers, or unsafe comparisons.

3. Maintainability

o When cryptographic requirements change (e.g., increasing PBKDF2 iterations), you only update one place.

4. Prevent Attacks

o Timing attacks: constantTimeEquals prevents attackers from guessing passwords based on comparison speed.

o Rainbow table attacks: generateSalt ensures identical passwords hash differently.

o Brute-force attacks: derivePassword with PBKDF2 slows down attackers.

_____

Example Usage

1. Password Hashing

```
char[] password = "SuperSecret123!".toCharArray();

byte[] salt = CryptoHelper.generateSalt();

byte[] hashedPassword = CryptoHelper.derivePassword(password, salt, 100_000, 256);


System.out.println("Salt (Base64): " + Base64.getEncoder().encodeToString(salt));

System.out.println("Hashed Password (Base64): " + Base64.getEncoder().encodeToString(hashedPassword));
```

- **generateSalt() → creates a random salt for this user.**
- **derivePassword() → hashes the password securely with PBKDF2.**
- **Even if two users have the same password, the salt ensures their hashes differ.**

_____

2. Session Token Generation

```
String token = CryptoHelper.generateSessionToken();

System.out.println("Session Token: " + token);


// Store the token hash instead of raw token

String tokenHash = CryptoHelper.sha256Base64(token);

System.out.println("Token Hash (stored): " + tokenHash);
```

- Generates a high-entropy token for sessions or API keys.
- Storing the hash instead of raw token prevents leaks if your database is compromised.

---

3. Constant-Time Comparison

```
byte[] hash1 = CryptoHelper.derivePassword(password, salt, 100_000, 256);

byte[] hash2 = CryptoHelper.derivePassword(password, salt, 100_000, 256);


if (CryptoHelper.constantTimeEquals(hash1, hash2)) {

   System.out.println("Passwords match!");

} else {

   System.out.println("Passwords do NOT match!");

}
```

- Using normal Arrays.equals() could leak timing information.
- constantTimeEquals ensures the comparison time does not vary based on input.

✅ **Summary**

A crypto helper in Java is essential for:

- Secure password storage
- Safe session management
- Prevention of timing and brute-force attacks
- Centralizing security logic for maintainability

Without a helper, you risk inconsistent security practices, which can lead to vulnerabilities.

Additional Evidence:

Examples of how to use putIfAbsent() method in java and the purpose of it

putIfAbsent() is a very handy method in Java, especially when working with **maps** like HashMap, ConcurrentHashMap, or LinkedHashMap.

□ Definition
putIfAbsent(key, value) is a method in the **Map interface** (since Java 8) that:
✅ **Inserts a value for a key only if that key is not already present in the map.**
If the key **already exists**, it **does not overwrite** the existing value.

□ Syntax
V putIfAbsent(K key, V value)

- **Parameters:**
  - key → the key to insert
  - value → the value to associate with the key
- **Returns:**
  - The *previous value* associated with the key, **or null** if there was none (meaning insertion happened).

🔍 **Example 1: Basic HashMap usage**

```
import java.util.HashMap;

public class Example1 {
   public static void main(String[] args) {
      HashMap<String, String> map = new HashMap<>();

      map.put("Alice", "Admin");
      map.put("Bob", "User");

      // This key does not exist yet, so it will be added
      map.putIfAbsent("Charlie", "Guest");

      // This key already exists, so it will NOT be overwritten
      map.putIfAbsent("Alice", "SuperAdmin");

      System.out.println(map);
   }
}
```

□ Output:

```
{Alice=Admin, Bob=User, Charlie=Guest}
```

Explanation:

- "Charlie" was absent → added successfully ✅
- "Alice" was already present → **no change** ❌

🔁 **Example 2: Using return value**

```
HashMap<String, Integer> scores = new HashMap<>();

Integer prev = scores.putIfAbsent("John", 90);
System.out.println(prev); // null (new entry added)

prev = scores.putIfAbsent("John", 100);
System.out.println(prev); // 90 (existing value returned)

System.out.println(scores);
```

□ Output:

```
null
90
{John=90}
```

- First call added "John" → 90 (no previous value)
- Second call didn't overwrite because "John" was already present

---

⚙️ **Example 3: Thread-safe map (ConcurrentHashMap)**
When using multi-threaded code, putIfAbsent() is **atomic** in ConcurrentHashMap.

```
import java.util.concurrent.ConcurrentHashMap;
```

```
public class Example3 {
   public static void main(String[] args) {
      ConcurrentHashMap<String, Integer> cache = new ConcurrentHashMap<>();

      cache.put("count", 10);

      // Safe even in multi-threaded environments
      cache.putIfAbsent("count", 20);
      cache.putIfAbsent("total", 50);

      System.out.println(cache);
   }
}
```

Output:

{count=10, total=50}

Here, even if multiple threads try to insert the same key at once, only the first one will succeed.

When to Use putIfAbsent()

Scenario	Why use it
Prevent overwriting existing data	You don't want to lose existing values
Build a cache	Only initialize entries once
Handle defaults safely	Insert default values if missing
Multi-threaded maps	Atomic operation avoids race conditions

In short

Method	Behavior
put(key, value)	Always inserts or replaces
putIfAbsent(key, value)	Inserts **only if key doesn't exist**