

Module 1 AI Code Analysis Foundations

PROMPT:

```
function processData(d) {  
  var r = [];  
  for(var i = 0; i < d.length; i++) {  
    if(d[i] != null) {  
      if(d[i].active == true) {  
        if(d[i].type == 'premium') {  
          var obj = {};  
          obj.id = d[i].id;  
          obj.name = d[i].firstName + ' ' + d[i].lastName;  
          obj.email = d[i].email;  
          obj.status = 'active';  
          r.push(obj);  
        } } }  
      }  
    }  
  }  
  return r;  
}
```

1. FIND ISSUES:

- Complexity problems
- Performance bottlenecks
- Security vulnerabilities
- Readability issues
- Potential bugs

2. MEASURE:

- Cyclomatic complexity
- Lines of code
- Number of responsibilities

AI Driven Development

3. SUGGEST:

- Specific improvements
- Refactoring opportunities
- Best practice violations

Provide concrete examples and fixes.

Cursor AI Answer:

Let's analyze the provided function step by step:

1. FIND ISSUES

a. Complexity Problems

- Deep Nesting: The function uses multiple nested `if` statements, making it harder to read and maintain.
- Single Responsibility Principle Violation: The function does filtering, mapping, and formatting in one loop.

b. Performance Bottlenecks

- Inefficient Looping: The function loops through the entire array, but this is typical for such filtering. No major performance bottleneck unless the array is huge.
- No Early Exit: For very large arrays, consider using more efficient array methods or breaking early if possible.

c. Security Vulnerabilities

- No Input Validation: The function assumes that `d` is always an array and that its elements have the expected properties. If not, it could throw errors.
- Data Leakage: If this function is used in a context where sensitive user data is handled, ensure that only necessary fields are exposed.

d. Readability Issues

- Variable Naming: `d` and `r` are not descriptive. Use meaningful names like `data` and `result`.
- Nested Ifs: Deep nesting reduces readability.
- Manual Object Construction: Could use destructuring or mapping for clarity.

e. Potential Bugs

- Null/Undefined Checks: Only checks for `d[i] !== null`, but doesn't check if `d[i]` is an object or if properties exist.
- String Concatenation: If `firstName` or `lastName` is missing, the result could be `"undefinedundefined"`.

2. MEASURE

a. Cyclomatic Complexity

- Cyclomatic Complexity: 4

(One for the function, plus one for each `if` statement: $1 + 3 = 4$)

b. Lines of Code

- Lines of Code: 15 (including braces and blank lines)

c. Number of Responsibilities

- Responsibilities: 3
 1. Filtering for non-null, active, premium users
 1. Formatting user data
 1. Returning the result array
-

3. SUGGEST

a. Specific Improvements

- Use Array Methods: Use `filter` and `map` for clarity and functional style.
- Input Validation: Check if `d` is an array.
- Destructuring: Use destructuring for cleaner code.
- Guard Clauses: Reduce nesting by returning early.

b. Refactoring Opportunities

Refactored Example:

javascript

Apply

```
function processData(data) {  
  if (!Array.isArray(data)) return [];  
  return data  
    .filter(  
      user =>  
        user &&  
        user.active === true &&
```

```

        user.type === 'premium'
    )

    .map(user => ({
        id: user.id,
        name: [user.firstName, user.lastName].filter(Boolean).join(' '),
        email: user.email,
        status: 'active'
    }));
}

```

c. Best Practice Violations

- Variable Naming: Use descriptive names.
- Avoid Deep Nesting: Use guard clauses or array methods.
- Input Validation: Always validate inputs.
- Avoid Magic Strings: Consider using constants for status/type if reused.

Summary Table

Issue Type	Problem/Observation	Suggested Fix/Example
Complexity	Deep nesting, multiple responsibilities	Use array methods, split responsibilities if needed
Performance	None significant for small arrays	For large data, consider streaming or chunking
Security	No input validation, possible data leakage	Validate input, only expose necessary fields
Readability	Poor variable names, deep nesting	Use descriptive names, guard clauses, array methods

Potential Bugs	Missing property checks, string concat with undefined	Use destructuring, default values, filter falsy names
Best Practices	Manual loops, magic strings	Use constants, array methods, input validation

In summary: Refactor to use array methods, validate input, use descriptive names, and avoid deep nesting. This will improve readability, maintainability, and robustness.

2. AI-Powered Code Health Audit

PROMPT:

Perform a comprehensive code health audit:

METRICS:

- Calculate complexity score for each function
- Identify functions over 20 lines
- Find duplicate code patterns
- Count nested conditions

ISSUES:

- Security vulnerabilities
- Performance bottlenecks
- Maintainability problems

AI Driven Development

- Testing gaps

REPORT:

- Prioritize issues by impact
- Suggest improvement roadmap
- Estimate effort for each fix

Format as a structured report with specific line numbers

Cursor AI Answer:

```

/*
=====
=====
Comprehensive Code Health Audit Report
File:
src/main/java/com/foodorder/usermanagement/service/impl/UserServiceImpl.java
va

```

1. METRICS & CODE ANALYSIS

A. Complexity Score (Cyclomatic Complexity) per Function:

- `getUserById` (lines 19-22): 2
- `getUserByEmail` (lines 24-27): 2
- `getAllUsers` (lines 29-31): 1
- `getAllSellers` (lines 33-35): 1
- `getAllCustomers` (lines 37-39): 1
- `updateUser` (lines 41-77): 13
- `deleteUser` (lines 79-84): 3
- `changePassword` (lines 86-92): 4

B. Functions Over 20 Lines:

- `updateUser` (lines 41-77): 37 lines

C. Duplicate Code Patterns:

- Null checks and field updates in `updateUser` (lines 45-74) are repeated for each field.
- Similar exception handling for "User not found" in `getUserById`, `getUserByEmail`, `deleteUser`.

D. Nested Conditions Count:

- `updateUser`: Up to 3 levels (e.g., `if (userDetails.isSeller()) { if (userDetails.getBusinessName() != null) ... }`)
- Other methods: 1 level

2. ISSUES IDENTIFIED

A. Security Vulnerabilities:

-
- Password update in updateUser (lines 56-58): No check if the user is allowed to change password, nor validation of password strength.
 - No authorization checks in any method (e.g., updateUser, deleteUser, changePassword) to ensure only permitted users can perform actions.
 - Potential for privilege escalation if userDetails.isSeller() is set to true by a non-admin (lines 63-74).

B. Performance Bottlenecks:

- No major performance issues detected; all repository calls are single operations.
- updateUser could be optimized by reducing unnecessary field updates.

C. Maintainability Problems:

- updateUser is too long (37 lines) and has repeated code for each field.
- No helper methods for field updates or seller-specific logic.
- No logging for critical operations (e.g., user deletion, password change).
- Exception messages are generic ("User not found") and do not specify the user id/email.

D. AI Driven Development - Testing Gaps:

- No unit tests or validation for input parameters (e.g., null checks for id, email).
- No test coverage for edge cases (e.g., updating with empty userDetails, invalid state transitions).
- No integration with audit logging or monitoring.

3. PRIORITIZED ISSUES BY IMPACT

1. ****Security: Lack of authorization checks**** (Critical, lines 41-92)
2. ****Security: Password update without validation**** (High, lines 56-58, 89-91)
3. ****Maintainability: Large, repetitive updateUser method**** (High, lines 41-77)

4. ****Maintainability: No logging for critical actions**** (Medium, lines 79-92)

5. ****Testing: No input validation or test coverage**** (Medium, all methods)

6. ****Performance: Minor, unnecessary field updates**** (Low, lines 41-77)

4. SUGGESTED IMPROVEMENT ROADMAP

A. Security Improvements

- Add authorization checks to all mutating methods (updateUser, deleteUser, changePassword).
- Validate password strength and restrict password changes to authorized users.
- Prevent privilege escalation by restricting isSeller and isEnabled updates to admins.

B. Maintainability Refactoring

- Refactor updateUser: Extract field update logic into helper methods or use a mapping utility.
- Add logging for user deletion, password changes, and failed operations.
- Improve exception messages to include user id/email.

C. Testing & Validation

- Add input validation for all public methods.
- Implement unit and integration tests for all service methods.
- Add audit logging for sensitive operations.

D. Performance Optimization

- Only update fields that have changed.
- Consider using patch/update DTOs for partial updates.

5. ESTIMATED EFFORT FOR EACH FIX

1. Add authorization checks: 2-3 hours
2. Password validation and restriction: 1-2 hours
3. Refactor updateUser (helper methods, reduce repetition): 2-3 hours
4. Add logging and improve exception messages: 1-2 hours
5. Add input validation and tests: 3-5 hours
6. Performance optimization (field change detection): 1 hour

6. SUMMARY TABLE (with line numbers)

Issue	Impact	Lines	Est.
Effort			
No authorization checks	Critical	41-92	2-3h
Password update w/o validation	High	56-58,89	1-2h
updateUser too long/repetitive	High	41-77	2-3h
No logging for critical actions	Medium	79-92	1-2h
No input validation/tests	Medium	all	3-5h
Minor performance inefficiency	Low	41-77	1h

END OF REPORT

*/

Documentation:

PROMPT:

Generate comprehensive documentation for this function:

updateUser
changePassword
getUserByEmail

REQUIREMENTS:

- Clear description of purpose
- Parameter types and descriptions
- Return value explanation
- Usage examples
- Error conditions
- Performance notes if relevant

FORMAT: Use JSDoc/PyDoc/etc appropriate for the language

Make it professional and helpful for other developer

Cursor AI Answer:

updateUser

```
/**
 * Updates the details of an existing user with the provided information.
 * <p>
 * Only non-null fields in {@code userDetails} will be updated. For seller accounts,
 * additional business-related fields can also be updated. The password will be
 * re-encoded if a new value is provided.
 * </p>
 *
 * @param id the unique identifier of the user to update (Long, not null)
 * @param userDetails a {@link User} object containing the new values for the user fields;
 * only non-null fields will be updated
 * @return the updated {@link User} entity after changes are persisted
 *
 * @throws ResourceNotFoundException if no user with the given {@code id} exists
 * @throws IllegalArgumentException if any provided field values are invalid
 *
 * <b>Usage Example:</b>
 * <pre>
 * User update = new User();
 * update.setFirstName("Alice");
 * update.setEmail("alice.new@email.com");
 * update.setPassword("newPassword123");
 * User updatedUser = userService.updateUser(5L, update);
 * </pre>
 *
 * <b>Error Conditions:</b>
```

```

* <ul>
* <li>User with the specified {@code id} does not exist</li>
* <li>Invalid or duplicate email (not currently checked, but recommended)</li>
* <li>Invalid password (e.g., empty string)</li>
* </ul>
*
* <b>Performance Notes:</b>
* <ul>
* <li>Performs a database lookup and update; performance is generally acceptable for single-user
updates.</li>
* <li>Method is transactional; all changes are rolled back if an exception occurs.</li>
* </ul>
*/

```

changePassword

```

/**
* Changes the password for the specified user after verifying the old password.
* <p>
* The old password must match the user's current password. The new password will be
* securely encoded before being saved.
* </p>
*
* @param id the unique identifier of the user whose password is to be changed (Long, not null)
* @param oldPassword the user's current password in plain text (String, not null)
* @param newPassword the new password to set, in plain text (String, not null)
*
* @throws ResourceNotFoundException if no user with the given {@code id} exists
* @throws IllegalArgumentException if the old password does not match the current password
*
* <b>Usage Example:</b>
* <pre>
* userService.changePassword(5L, "oldPass123", "newSecurePass456");
* </pre>
*
* <b>Error Conditions:</b>
* <ul>
* <li>User with the specified {@code id} does not exist</li>
* <li>Old password does not match the user's current password</li>
* <li>New password is empty or invalid (not explicitly checked, but recommended)</li>
* </ul>
*
* <b>Performance Notes:</b>
* <ul>
* <li>Performs a password hash comparison and a database update.</li>
* <li>Method is transactional; changes are rolled back if an exception occurs.</li>
* </ul>
*/

```

getUserByEmail

```
/**
 * Retrieves a user entity by their email address.
 *
 * @param email the email address of the user to retrieve (String, not null)
 * @return the {@link User} entity associated with the given email
 *
 * @throws ResourceNotFoundException if no user with the specified email exists
 *
 * <b>Usage Example:</b>
 * <pre>
 * User user = userService.getUserByEmail("alice@email.com");
 * </pre>
 *
 * <b>Error Conditions:</b>
 * <ul>
 * <li>No user exists with the specified email address</li>
 * </ul>
 *
 * <b>Performance Notes:</b>
 * <ul>
 * <li>Performs a database lookup by email; performance is optimal if the email field is indexed.</li>
 * </ul>
 */
```