Diagnostic Analysis Report

Customer Problem

Whenever I try to send a payload to the endpoint I created, the micro integrator gives an error

Log Analysis

```
Summary of Findings:
The logs show that while the HealthcareAPI repeatedly logs a welcome message, a JSON parsing error occurs at 2025-05-05 14:34:19,414.
The error message indicates that the system encountered an illegal character (“<d”) while processing a payload expected to be in JSON format.
The exception is thrown during the conversion or reading of JSON data, as seen in the stack trace from the JsonXMLStreamReader and the JsonUtil classes.
There is a clear indication that the payload being sent might contain unexpected or malformed content, possibly due to incorrect formatting or a misconfiguration in the endpoint.
No other critical errors or component failures are evident, and the normal log messages resume after the error, suggesting an isolated issue.

SUSPECTED_CLASSES: [{'package': 'org.apache.synapse.commons.json', 'class': 'JsonUtil', 'issue_line': 921}, {'package': 'org.apache.synapse.commons.json', 'class': 'JsonUtil', 'issue_line': 1042}, {'package': 'org.apache.synapse.commons.staxon.core.json', 'class': 'JsonXMLStreamReader', 'issue_line': 88}, {'package': 'org.apache.synapse.commons.json', 'class': 'JsonDataSource', 'issue_line': 154}, {'package': 'org.apache.synapse.mediators.transform', 'class': 'PayloadFactoryMediator', 'issue_line': 145}]

ERROR_MESSAGE: "Illegal character: <d>"
```

Comprehensive Thread Analysis

Not applicable - no problematic threads identified.

Class Files Analysis

Below is a detailed analysis of the JsonUtil class with respect to the customer problem and potential issues. The analysis covers design, error handling, resource management, concurrency, and code■smells. Each point is followed by suggestions for improvement.

1. Design Flaws

- The class handles several conversion modes (XML to JSON and vice■versa) in a single, large class. The multitude of overloaded methods (for example, getNewJsonPayload with InputStream, String, and byte[] parameters) increases complexity and maintenance burden.
- There is a mix of deprecated methods and new implementations. The deprecated methods remain in the code (for example, newJsonPayload and its variants), which may confuse new developers and increase the risk of inconsistent behavior.
- The configuration of JSON/XML conversion is done via static initialization blocks. This “hardwiring” of configuration might complicate testing, reuse, and changes at runtime.
- The use of a custom ReadOnlyBIS wrapper class for an input stream is not clearly justified. While it aims to provide a non■closable, re■readable stream, it introduces nonstandard behavior that could make the code harder to debug.

Suggested Improvements:
- Split the responsibilities into smaller classes (for example, separate builders, converters, and stream wrappers).
- Remove or clearly document deprecated methods and provide a well■defined API boundary for external users.
- Consider dependency injecting configuration data rather than relying on static properties.
- Revisit the design of ReadOnlyBIS to either use a well■known reusable stream solution or add extensive documentation describing its behavior.

2. Error Handling Issues

- Many methods (for example, writeAsJson and getNewJsonPayload) catch exceptions and rethrow them as AxisFault while logging minimal context information.
- In the getNewJsonPayload method that produces a new OMElement from a JSON string (line 1042 – the reported problematic line), the code appears to have a commented out portion. It is unclear whether the commented code was meant to be active.
- In exception blocks handling IO and XMLStreamException errors, there is a tendency to simply log the error message without additional recovery or context.
- The style of exception handling sometimes uses logging.error and then immediately throwing an exception, without cleaning up resources or ensuring the state is consistent.

Suggested Improvements:
- Improve error messages by including more contextual details (e.g. which payload or which conversion mode encountered the error) so that debugging “Illegal character: <d>” becomes easier.
- Establish a consistent error handling strategy; consider wrapping lower-level exceptions in meaningful domain-specific exceptions.
- Verify that all resources (such as streams and readers) are closed or reset in finally blocks or via try-with-resources where possible.

3. Resource Management Problems

- Several conversion methods create streams (for example, ByteArrayInputStream and ByteArrayOutputStream) and rely on IOUtils.copy or manual flush/close handling.
- In the writeJsonStream method, the JSON stream is not explicitly closed if an exception occurs during copying.
- The custom ReadOnlyBIS class overrides close to perform a reset rather than actually freeing resources. This behavior, while intentional, could confuse clients expecting the stream to be closed.
- Mark and reset support of streams is assumed without a check or fallback strategy. If the underlying stream does not support mark/reset appropriately, the conversion may fail.

Suggested Improvements:
- Where possible, use try-with-resources to handle streams so that they are automatically closed even if an exception occurs.
- Revisit the design of ReadOnlyBIS to ensure its behavior is well documented; consider exposing a method to “really” close a stream if required.
- In any conversion method that creates temporary streams (such as ByteArrayOutputStream), ensure that flush and closing are done inside a finally block.

4. Threading and Concurrency Issues

- The class mostly uses static factories and configuration objects that are initialized once. These static members are immutable (or effectively immutable) during runtime so there appears to be no direct issue with concurrent access.
- However, the reuse of shared configuration objects (like jsonOutputFactory and jsonInputFactory) must be thread■safe. If any of the underlying factories are stateful, then concurrent requests might interfere with one another.
- The logger usage is thread■safe; however, any mutable shared state in the message context or custom stream wrappers should be carefully reviewed for thread-safety.

Suggested Improvements:
- Verify that the underlying JSON/XML input/output factories are thread-safe. If not, consider instantiating a new one per request or using thread-local holders.
- Avoid any static mutable state. Ensure that any caching (such as in cachedCopyOfJsonPayload) correctly synchronizes access if multiple threads may access the same message context concurrently.

5. Specific Code Smells and the Reported “Illegal character: <d>” Issue

- At line 1042, the commented-out code hints at a potential issue in converting a JSON string to a new JSON payload using new ByteArrayInputStream(jsonString.getBytes()). The default charset is used here, which might lead to encoding problems if the actual JSON payload uses a different charset. This could result in illegal characters (such as a stray “<d>”) being interpreted during the conversion.
- The error “Illegal character: <d>” may also indicate that non-JSON characters (possibly XML or HTML fragments) are being fed into the JSON conversion routines. The method to detect an empty or invalid payload checks for certain characters, but the logic may not be robust for all cases.

Suggested Improvements:
- Use explicit character encoding when converting the JSON string to bytes (for example, jsonString.getBytes(StandardCharsets.UTF_8)) to prevent misinterpretation.
- Enhance input validation logic in getNewJsonPayload to verify that the stream truly contains valid JSON (for example, by using a JSON parser in a try-catch block before proceeding).
- Add unit tests for JSON payload conversion that include malformed JSON input

(including inputs containing characters such as “<d>”) so that the failure can be reproduced and handled gracefully.

6. General Maintenance Suggestions

- Refactor long methods into smaller helper methods. For example, methods that remove indentations and remove namespaces can be placed in a separate utility class.
- Comment the rationale behind certain design decisions (such as the use of a custom stream wrapper) so future maintainers can understand the trade-offs.
- Standardize logging formats and error messages so that all thrown exceptions include the message context’s ID and relevant payload information.

Conclusion

This analysis reveals that JsonUtil shows several areas for improvement in design, error handling, and resource management. The probable cause of the “Illegal character: <d>” error appears to be related to encoding issues and inadequate verification of input payloads. Adopting explicit charset handling, refactoring methods for clarity, and improving error handling and resource management will result in a more robust and maintainable codebase.

Conclusions and Recommendations

Diagnosis:
The error "Illegal character: <d>" is caused by feeding malformed or mis-encoded payload data into the JSON conversion routines. The payload likely contains unexpected characters—possibly remnants of XML/HTML or misinterpreted byte sequences due to the use of the system's default charset instead of an explicit encoding. The conversion code in the JsonUtil class, especially at line 1042, does not enforce strict input validation or character encoding, and the presence of commented-out code hints at unresolved issues around proper byte conversion and payload verification.

Actionable Steps:
1. Modify the payload conversion logic to explicitly use a defined charset (e.g., UTF-8) when converting the JSON string to bytes.
2. Enhance input validation in getNewJsonPayload by verifying the input payload structure and ensuring it conforms to valid JSON before processing.
3. Refactor the conversion routines by breaking down large methods into smaller, testable helpers that handle specific responsibilities such as formatting, encoding, and resource management.
4. Remove or clearly document deprecated methods and the custom ReadOnlyBIS behavior to avoid ambiguity in stream handling.
5. Implement comprehensive unit tests that cover edge cases, including malformed JSON or payloads with extraneous characters (e.g., "<d"), to ensure the correction fixes similar issues in the future.
6. Review and improve error handling to include detailed context, ensuring that any exceptions capture relevant details about the payload and conversion mode to facilitate debugging.