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

```
Analysis Summary:
The logs show repeated successful log messages with "Welcome to HealthcareService"
from the HealthcareAPI at the /healthcare/querydoctor endpoint. An error occurs at
14:42:06,820 where the system fails to parse the payload. The error "Illegal
character: <d>" is thrown during JSON to XML conversion. The stack trace indicates
that the failure occurs while processing the JSON payload into an XML structure using
components from the JSON utilities and staxon modules. The exception occurs in the
context of the PayloadFactoryMediator, suggesting that the payload received by the
endpoint may contain malformed JSON or unexpected characters. There is a pattern where
the error only appears when the payload is sent, while routine log messages continue
otherwise. This points to a possible issue with payload formatting or a conflict in
the JSON to XML parsing logic.

SUSPECTED_CLASSES: [{'package': 'org.apache.synapse.commons.json', 'class':
'JsonUtil', 'issue_line': 942}, {'package':
'org.apache.synapse.commons.staxon.core.json', 'class': 'JsonXMLStreamReader',
'issue_line': 88}, {'package':
'org.apache.synapse.commons.staxon.core.json.stream.impl', 'class': 'JsonScanner',
'issue_line': 777}, {'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 the detailed analysis and recommendations:

1. Design Issues and Code Organization
The JsonUtil class is very large with lengthy static methods that mix concerns such as JSON–XML conversion, stream management, and legacy payload handling. This monolithic design makes it difficult to maintain and extend. In addition, the use of multiple “generate…Factory” static methods tied to global properties (and potential duplication in generateJSONOutputFactory and generateJSONOutputFactoryWithOveride) suggests that the configuration aspects should be encapsulated into dedicated configuration classes with a clean builder pattern.

2. Error Handling Problems
There are several areas where exceptions are caught and logged but then either rethrown as AxisFaults or swallowed. For example, in the removeChildrenFromPayloadBody method the commented out invocation for removeIndentations(body) (noted by the “line 942” comment) indicates a potential failure point. When an exception occurs here, the code detaches the first element and then throws a SynapseCommonsException. More consistent and informative error handling is needed. In multiple places, IOException and XMLStreamException are caught and wrapped but without restoring the original context, making debugging difficult. In addition, errors during stream copy operations or reset failures in writeJsonStream may propagate incomplete conversions.

3. Resource Management and Stream Handling
The class makes extensive use of InputStreams and Readers without always ensuring that resources are closed properly. The use of IOUtils.copy and IOUtils.closeQuietly is helpful; however, in certain branches (for example inside convertOMElementToJson) the XMLStreamReader and XMLEventWriter are closed in finally blocks but the underlying streams may not be closed or reset properly.
The custom ReadOnlyBIS class overrides close() to perform a reset rather than a true close, which may leave underlying streams open if the caller expects that closing the stream frees system resources. This design is problematic if the stream is used by multiple threads or passed to other components.

4. Concurrency Considerations
The use of static variables (such as the jsonOutputFactory and jsonInputFactory) means that configuration is shared among all threads. Although these factories appear to be thread■safe, care must be taken when overriding configuration later via generateJSONOutputFactoryWithOveride. In addition, the ReadOnlyBIS inner class does not support marking in a thread-safe manner; concurrent usage could lead to unexpected resets or data corruption.

5. Potential Encoding and Illegal Character Issues
The reported log error “Illegal character: <d>” could result from incorrect encoding conversions when writing the JSON stream. The writeJsonStream method inspects the outbound character set encoding and may use a conversion that is vulnerable if the inbound encoding is not reliably determined. For instance, using new String(inboundBuffer).getBytes(outboundCharsetEncoding) may be error prone if the default charset is not appropriate. There is also complexity when comparing the inboundCharsetEncoding with the default Charset – subtle differences may lead to unexpected characters being output. A more robust approach would be to use explicit character decoding and encoding using Charset objects rather than relying on default conversions.

6. Code Smells and Anti■patterns
The class contains several deprecated methods (e.g. newJsonPayload overloads) that can confuse users of the API. These should be clearly documented and eventually removed to cut down on redundancy. Also, some methods use “magic” values such as the QName JSON_OBJECT with no clear explanation, and the configuration of namespaceSeparator using the hardcoded value '\\u0D89' may not be self–explanatory. Finally, many try/catch blocks log errors without properly rethrowing exceptions or cleaning up state, which could hide failures in production.

7. Recommended Improvements and Fixes
• Refactor the JsonUtil class into smaller, more focused classes or modules. For example, separate JSON–XML conversion logic from stream handling and configuration management.
• Replace global static factory instances with dependency–injected versions or instance–level configuration to improve testability and reduce threading risk.
• Improve error handling by including the original exception stack trace and context data, and avoid swallowing exceptions. Ensure that all resources are closed or reset explicitly using try–with–resources where feasible.
• Revisit the custom ReadOnlyBIS design to decouple “read-only” behavior from resource management. Consider wrapping the InputStream in a decorator that supports both reset and proper resource finalization.
• Rethink the encoding conversion logic in writeJsonStream by using java.nio.charset.Charset and proper decoding/encoding streams so that illegal characters such as “<d>” are either handled or rejected with a clear error message.
• Remove deprecated APIs and document the usage expectations more clearly. Document the behavior of factory methods and validate that the properties are consistently applied.
• Add more comprehensive logging (using consistent log levels) to capture the context when an exception occurs in JSON conversion or stream handling.
• Address the commented out call at line 942 (removeIndentations(body)). Verify why it was disabled and whether re-enabling it may fix or cause additional problems.

By implementing these improvements, the code will become more maintainable, robust against malformed input and resource leaks, and more predictable under concurrent usage. The potential “Illegal character: <d>” error may be resolved by ensuring proper encoding conversions and stricter checks on the JSON payload content.

Conclusions and Recommendations

Diagnosis:
The error is caused by malformed JSON content or an encoding conversion issue during the JSON to XML transformation. The illegal character "<d>" indicates that the payload contains unexpected characters that the conversion logic in the JsonUtil and associated classes cannot handle. The conversion process in the PayloadFactoryMediator, relying on the JSON utilities and staxon modules, encounters inconsistent handling of character encoding and stream management. Additionally, the monolithic architecture and error handling inconsistencies in these classes add to the fragility of the payload processing, leading to parsing failures.

Actionable Steps:
1. Validate the JSON payload by ensuring it meets proper syntax standards and does not contain extraneous or malformed characters before sending it to the endpoint.
2. Introduce stricter input validation on the micro integrator to check for illegal characters and enforce correct encoding.
3. Refactor the JSON to XML conversion code by separating concerns into smaller, focused modules, reducing reliance on monolithic static methods.
4. Update the encoding conversion logic in the writeJsonStream method to use java.nio.charset.Charset objects for explicit decoding and encoding, ensuring consistent behavior regardless of default charsets.

5. Improve error handling by encapsulating and logging the full context and original stack traces, avoiding suppression of exceptions.
6. Revisit and update the resource management approach by using try-with-resources or explicit closing of streams to avoid issues related to the custom ReadOnlyBIS design.
7. Remove deprecated APIs and redundant methods while clearly documenting the expected configuration and usage patterns of the JSON conversion facilities.
8. Thoroughly test the complete end-to-end flow with various payloads to ensure that encoding conversions and stream resets are handled correctly and that illegal characters are appropriately managed.