ChatGPT

Prompt for Rewriting QuickFIX/J as a Node.js Library

QuickFIX/J is a **full-featured, open-source Java FIX engine** that supports FIX versions 4.0 through 5.0SP2 (FIXT1.1) [1]. The new Node.js package must **replicate its complete feature set**: session management with logon/logout/heartbeat/test-request logic, message parsing/validation, initiator/acceptor transports, configurable message stores (file and memory), flexible logging, error handling, and an XML-based FIX Data Dictionary. It should be modular (e.g. `/core`, `/transport`, `/fix`, `/store`, `/config`, `/logging` directories), use asynchronous I/O via Node's `net` and `tls` modules (similar to QuickFIX/J's Java NIO usage [2]), and export intuitive APIs for building FIX Initiators and Acceptors. Support for multiple FIX versions (4.0, 4.2, 4.4, 5.0, FIXT1.1) [1] is required. The code will be plain JavaScript (ES6+), but structured so users can adopt TypeScript if desired. Include a buildable `package.json`, comprehensive `README.md`, and usage examples.

Key Requirements and Context: QuickFIX/J uses a session settings file (with `[DEFAULT]` and `[SESSION]` sections) parsed by a `SessionSettings` class [3]. Our Node.js version should similarly use a configuration-driven approach (e.g. parsing a QuickFIX-style `.cfg` or JSON file into a Settings object). QuickFIX/J's built-in message stores include **file-based and in-memory stores** [4]; the Node.js rewrite must at least support these (with a factory to create stores per session). Logging in QuickFIX/J is pluggable (file, console, JDBC, etc.) [5]; the new library should abstract logging so developers can plug in different transports (console, file, or external logging libraries). QuickFIX/J loads **XML Data Dictionaries** (FIX40.xml, FIX42.xml, etc. included by default) for field/message definitions [6]; we will convert those dictionaries to JSON and load them as metadata to drive parsing and validation.

Feature Breakdown

- **FIX Version Support:** Fully support FIX 4.0, 4.2, 4.4, 5.0 and FIXT1.1. Use separate dictionary files or metadata for each version (converted from QuickFIX/J's XML dictionaries) [1] [6]. The engine should detect the FIX version per session (from `BeginString` / `DefaultApplVerID` in settings) and load the appropriate definitions.

- **Session Management:** Implement a Session class that tracks sequence numbers, connection state, heartbeat intervals, and performs the FIX logon/logoff handshake. Handle automatic heartbeats and test requests, resend logic on gaps, sequence resets, and logout timeout. Mirror QuickFIX/J's session lifecycle callbacks (`onCreate`, `onLogon`, `onLogout`) and ensure sessions exist regardless of current connection [7].

- **Message Parsing and Generation:** Provide parsers to convert raw FIX message strings (tag=value pairs) into JavaScript objects (with nested groups) and vice versa. Also allow message construction from a JSON-friendly structure (mapping tag numbers or field names to values). Use the loaded FIX Data Dictionary metadata to validate fields, check required fields, and encode/decode repeating groups.

- **Transport Layer (Initiator/Acceptor):** Create separate modules for initiating connections and accepting incoming connections. For **initiators**, use `net.Socket` or `tls.connect()` (with

SSL options) to connect to counterparts; for **acceptors**, use `net.createServer()` or `tls.createServer()` to listen for connections. Each connection should be associated with a Session based on session identifiers. Ensure asynchronous, event-driven networking (following Node.js best practices).

- **Message Store Abstractions:** Define an interface for message stores with methods like `storeMessage(sessionID, message)`, `getMessage(sessionID, seqNum)`, `getNextSenderMsgSeqNum()`, `getNextTargetMsgSeqNum()`, `reset()`, etc. Provide at least two implementations: **FileStore** (persist messages in files, one per session, emulating QuickFIX/J's file format) and **MemoryStore** (keep messages in-memory for testing or ephemeral use). A `StoreFactory` should pick the store type based on configuration (e.g. `FileStoreFactory` vs. `MemoryStoreFactory`).

- **Logging System:** Architect a pluggable logging layer. Include at minimum a `FileLog` and a `ConsoleLog`. Each log entry should record timestamps, session info, message direction (IN/OUT), and the raw FIX string. Optionally allow integration with common log libraries (e.g. Winston or Bunyan) via a `LogFactory` abstraction. Support configurable log destinations per session in settings.

- **Configuration System:** Support loading settings from a configuration file (QuickFIX-style or JSON). At minimum, parse a QuickFIX `.cfg` (INI) file similar to QuickFIX/J: use sections `[DEFAULT]` and `[SESSION]` [8]. Build a `SessionSettings` class that reads this file/JSON and constructs settings objects (BeginString, SenderCompID, TargetCompID, ConnectionType, HeartBtInt, SSL parameters, etc.). Use these settings to initialize sessions, transports, stores, and loggers.

- **Data Dictionary Handling:** Convert QuickFIX/J's XML FIX dictionaries (e.g. `FIX50.xml`, `FIX44.xml`) into clean JSON files. Create a `DataDictionary` module that loads the JSON dictionary for a given FIX version, providing field definitions (names, types, required flags), message structures, valid values/enums, and repeating-group layouts. Use this metadata to drive message parsing/validation.

- **Application Callbacks/Event Handling:** Allow users to define custom handlers for inbound and outbound messages. Similar to QuickFIX/J's `Application` interface, the library should support callbacks like `onCreate(sessionID)`, `onLogon(sessionID)`, `onLogout(sessionID)`, `fromAdmin(message, sessionID)`, `toAdmin(message, sessionID)`, `fromApp(message, sessionID)`, and `toApp(message, sessionID)`. In Node.js, this could be modeled via an `EventEmitter` or by passing a callback object when creating an Initiator/Acceptor. Ensure all callbacks are invoked asynchronously. Example usage:

```
const app = {
  onCreate: (sessionID) => { /* ... */ },
  onLogon:   (sessionID) => { /* ... */ },
  fromApp:   (msg, sessionID) => { /* handle incoming application-level
message */ },
  // ...
};
```

```
const acceptor = new SocketAcceptor(app, sessionSettings);
acceptor.start();
```

- **Testing and Quality:** Plan for comprehensive unit and integration tests. For each module (core engine, parser, transport, store, config, etc.), write tests (e.g. using Mocha or Jest) to verify behavior: correct message encoding/decoding, sequence number handling, session timeouts, logon/logout flows, store persistence, etc. Include an integration test that spins up an Acceptor and Initiator to establish a FIX session (logon sequence), exchange a few messages, and logout cleanly. Strive for high coverage.

- **Extensibility:** Design so users can add custom FIX fields or message types. For example, allow loading additional user-defined dictionary entries (maybe via a custom JSON file) and merging them into the DataDictionary at runtime. Expose APIs or hooks for custom session-level behavior (e.g. applying business rules in callbacks). Make the code modular so one can plug in alternative transports or storage without altering core logic.

Modules and File Structure

Organize the code into logical directories and files. For example:

- **/core** (the engine and FIX primitives)
- `SessionSettings.js` – parses config files (INI/JSON) into settings.
- `SessionID.js` – represents the unique session identifier (BeginString/Sender/Target).
- `Session.js` – main session class (sequence state, send/receive logic, heartbeat timer).
- `Engine.js` – orchestrates multiple sessions; routes incoming data to correct session.
- `Message.js`, `Field.js`, `FieldMap.js`, `Group.js` – classes for FIX messages, fields, and repeating groups.
- `MessageParser.js` – functions to parse raw FIX strings into `Message` objects using the DataDictionary.
- `MessageBuilder.js` – functions to serialize `Message` objects back to FIX strings.
- `DataDictionary.js` – loads the JSON dictionary for a given FIX version.
- `MessageFactory.js` – (optional) create typed message instances if using code-generated classes.

- `Application.js` – an abstract/base class or interface for user callbacks (onCreate, onLogon, etc.).

- **/transport** (network I/O)

- `SocketInitiator.js` – connects to a FIX server using `net.Socket`; manages reconnection scheduling.
- `SocketAcceptor.js` – listens for FIX client connections using `net.createServer`.
- `TLSInitiator.js` / `TLSAcceptor.js` – same as above but with SSL/TLS (`tls.connect`, `tls.createServer`).

- These should each extend a common base that attaches data handlers to raw sockets and ties into Sessions.

- **/store** (message persistence)

- `MessageStore.js` (interface) – defines methods for storing/retrieving messages and seq nums.
- `FileStore.js` – implements `MessageStore`, saving messages in text files (append-only per session).
- `MemoryStore.js` – in-memory store (for testing).
- `StoreFactory.js` – chooses store type from settings (e.g. `FileStoreFactory`, `MemoryStoreFactory`).

- **/logging**

- `LogFactory.js` – creates logger instances based on config (console, file, etc.).
- `ConsoleLog.js` – simple console logger.
- `FileLog.js` – appends logs to file.

- Optionally allow integration with popular loggers.

- **/config**

- `ConfigParser.js` – reads and parses the QuickFIX-style `.cfg` (or JSON) and produces a `SessionSettings` object [3].
- Store configuration options (e.g. `settings.connectionType = "acceptor"`, heartbeat interval, SSL cert paths, etc.).

- **/fix/data**

- JSON files for each FIX version's dictionary (converted from XML). Example: `FIX40.json`, `FIX42.json`, `FIX44.json`, `FIX50SP2.json`, `FIXT11.json`, etc.

- **tests/**

- Organize tests mirroring the code structure (e.g. `tests/core/Session.test.js`, `tests/transport/SocketInitiator.test.js`, etc.).
- Include end-to-end tests connecting a live initiator and acceptor.

Mapping Java Classes to JavaScript Modules

To ensure parity with QuickFIX/J, explicitly map key Java classes/interfaces to JS modules:

- `quickfix.SessionSettings` → **core/SessionSettings.js** (handles `[DEFAULT]` / `[SESSION]` config parsing) [8].
- `quickfix.SessionID` → **core/SessionID.js** (represents sender/target comp IDs + qualifier).
- `quickfix.Session` → **core/Session.js** (session state, nextSeqNums, logout logic).
- `quickfix.Message` → **core/Message.js** (wrapper for tag/value pairs).
- `quickfix.FieldMap` / `quickfix.Group` → **core/FieldMap.js**, **core/Group.js** (handle nested repeating groups).
- `quickfix.MessageStoreFactory` → **store/StoreFactory.js** (factory to create file or memory stores).

- `quickfix.FileStore` → **store/FileStore.js**; `quickfix.MemoryStore` → **store/MemoryStore.js**.
- `quickfix.LogFactory` → **logging/LogFactory.js**; `quickfix.FileLog` → **logging/FileLog.js**; `quickfix.ScreenLog` or console → **logging/ConsoleLog.js**.
- `quickfix.Acceptor` → **transport/SocketAcceptor.js** (and **TLSAcceptor.js**).
- `quickfix.Initiator` → **transport/SocketInitiator.js** (and **TLSInitiator.js**).
- `quickfix.Application` (interface) → **core/Application.js** (define or document callback methods). Possibly also an `ApplicationAdapter` base class with no-op implementations.

Include any utility classes (e.g. `MessageUtil.sendToTarget`) as methods on the `Session` or in `MessageBuilder`.

Parsing XML Data Dictionaries

QuickFIX/J uses XML FIX data dictionaries (e.g. `FIX44.xml`) [6] to drive parsing and validation. In the Node.js version:

- **Dictionary Conversion:** Pre-process the official QuickFIX/J XML dictionary files into JSON (e.g. using a build script or manual conversion). Each JSON should list fields (tag, name, type, enumerated values) and messages (with required/optional fields and group definitions).
- **Loading and Validation:** Implement a `DataDictionary.js` module that loads the appropriate JSON dictionary based on the session's FIX version. The `MessageParser` should refer to this dictionary to validate incoming messages: check for required fields, valid data types, and correct group structures. When constructing messages, use the dictionary to ensure fields are allowed in the given message type.
- **Versioning:** For FIXT1.1 (Transport Independence), handle `DefaultApplVerID` to load the correct application-level dictionary (FIX 5.0, etc.) in addition to transport version (FIXT.1.1).

By using JSON, parsing is faster and easier in JavaScript, and users can even edit the dictionaries if needed.

Event Handling and Callbacks

Model session and message events so applications can hook into them:

- **Session Events:** Emit or callback on events like `create`, `logon`, `logout`, and `disconnect`. For example, when a TCP connection is accepted and a new session is initialized, call `application.onCreate(sessionID)`. On successful logon handshake, call `application.onLogon(sessionID)`; on logout or lost connection, call `application.onLogout(sessionID)`. Ensure these are asynchronous (use `setImmediate` or Promises) to avoid blocking the engine.
- **Message Events:** When a new message arrives, first process administrative-level messages (`Logon`, `Heartbeat`, `TestRequest`, etc.), then invoke `application.fromApp(message, sessionID)` for application-level messages. Before sending any message, invoke `application.toAdmin` / `toApp` as appropriate (allow the app to modify or reject messages).
- **API Design:** One approach is to allow the user to pass an object implementing these callbacks to the `Initiator` or `Acceptor`. Alternatively, each `Session` could be an `EventEmitter` emitting named events (e.g. `'logon'`, `'message'`). In either case, document clearly how a user application should plug in its logic.

Configuration and Logging Abstractions

- **Configuration:** Provide a `SessionSettings` (or similar) class that reads a config file. It should support QuickFIX-style `.cfg` (INI) format with `[DEFAULT]` and `[SESSION]` [8], as well as possibly a JSON format. Store settings such as `ConnectionType`, `HeartbeatInterval`, SSL certificates, file paths for store/log, etc. Ensure any missing required setting throws a descriptive error (similar to QuickFIX/J's `ConfigError`).
- **Logging:** Use a `LogFactory` to create logger instances for each session. The factory reads config (e.g. `FileLogPath`, `LogLevel`) and returns a logger. The logger interface should have methods like `onIncoming(messageString)`, `onOutgoing(messageString)`, and `onEvent(infoString)`. By default, implement a file logger (appends messages to a log file per session) and a console logger (prints to stdout). Allow combining (e.g. file + console) if desired.

For example:

```
const logFactory = new LogFactory(settings);
const logger = logFactory.create(sessionID);
logger.onIncoming("8=FIX.4.2\u00019=...");
```

All logs should include timestamps and direction (IN/OUT) for traceability.

Testing Strategy

- **Unit Tests:** For each module, write unit tests covering normal and edge cases. Examples: parsing a valid FIX message string and reconstructing it; throwing errors on invalid messages; storing and retrieving messages in FileStore; constructing sessions with various settings.
- **Integration Tests:** Simulate a full session: start a `SocketAcceptor` and a `SocketInitiator` with matching settings, verify they exchange logon and logout successfully, and allow the application to send a few FIX messages in between. Test reconnection and gap-fill behavior by artificially dropping sequences.
- **Test Coverage:** Aim for high coverage (all major flows and error paths). Tests should not require a real FIX counterparty; they should use the library itself for both sides. Use a Node test framework like Mocha or Jest.

Extensibility

Design the library so developers can extend it:

- **Custom Message Types:** Allow loading additional dictionary entries (e.g. custom FIX fields/messages) by providing an extra JSON dictionary or modifying the loaded dictionary at runtime.
- **Pluggable Components:** Since stores and loggers use factory patterns, users should be able to implement and configure their own `MessageStore` or `Log` by extending abstract classes.
- **TypeScript Upgradability:** While written in JS, organize code with ES6 classes and clear interfaces so a TypeScript port would be straightforward. Document public APIs (using JSDoc) to assist in type annotation.
- **Configuration Hooks:** Permit overriding default behavior (e.g. a custom session schedule or automated re-logon logic) via callbacks or subclassing.

By following this detailed prompt, an AI code generator should produce a clean, modular Node.js library mirroring QuickFIX/J's capabilities, with each file and module clearly defined and documented.

Sources: QuickFIX/J feature list and documentation for FIX versions, session settings, and data dictionaries [1] [3] [6], as well as feature summaries (message store, logging) [5], informed this specification.

---

[1] [7] GitHub - quickfix-j/quickfixj: QuickFIX/J is a full featured messaging engine for the FIX protocol. - This is the official project repository.
https://github.com/quickfix-j/quickfixj

[2] [4] [5] QuickFIX/J - Free, Open Source Java FIX engine
https://www.quickfixj.org/

[3] [6] [8] Configuring QuickFIX/J
https://www.quickfixj.org/usermanual/2.3.0/usage/configuration.html