QuantumLedger™ — A High-Performance Event-Sourced Java Ledger System (MAANG-Level Project)

```
import java.lang.annotation.*;

import java.lang.reflect.*;

import java.math.BigDecimal;

import java.time.LocalDateTime;

import java.util.*;

import java.util.concurrent.*;

import java.util.function.*;

import java.util.logging.*;

import java.util.stream.*;


// ---------------------------------------------

// Annotations & Reflection-based DI Injection

// ---------------------------------------------

@Retention(RetentionPolicy.RUNTIME)

@Target(ElementType.FIELD)

@interface Inject {}


class DependencyInjector {

    public static void inject(Object obj) throws IllegalAccessException {
        Class<?> clazz = obj.getClass();
        for (Field field : clazz.getDeclaredFields()) {
            if (field.isAnnotationPresent(Inject.class)) {
                field.setAccessible(true);
                Object instance = SingletonRegistry.getInstance(field.getType());
                field.set(obj, instance);
            }
        }
    }
}

// ---------------------------------------------
// SingletonRegistry for dependency management
// ---------------------------------------------
class SingletonRegistry {
    private static final Map<Class<?>, Object> instances = new ConcurrentHashMap<>();

    public static <T> T getInstance(Class<T> clazz) {
        return clazz.cast(instances.computeIfAbsent(clazz, SingletonRegistry::instantiate));
    }

    private static <T> T instantiate(Class<T> clazz) {
        try {
            T instance = clazz.getDeclaredConstructor().newInstance();
            DependencyInjector.inject(instance);
            return instance;
        } catch (Exception e) {
            throw new RuntimeException("Failed to instantiate: " + clazz.getName(), e);
        }
    }
}

// ---------------------------------------------
// Immutable Domain Model using Builder Pattern
// ---------------------------------------------
final class LedgerEntry {
    private final String id;
    private final LocalDateTime timestamp;
    private final String account;
    private final BigDecimal amount;
    private final String type; // debit/credit
    private final Map<String, String> metadata;

    private LedgerEntry(Builder builder) {
        this.id = builder.id;
        this.timestamp = builder.timestamp;
        this.account = builder.account;
        this.amount = builder.amount;
        this.type = builder.type;
        this.metadata = Collections.unmodifiableMap(new HashMap<>(builder.metadata));
```

```
}

public static class Builder {
    private final String id = UUID.randomUUID().toString();
    private LocalDateTime timestamp = LocalDateTime.now();
    private String account;
    private BigDecimal amount;
    private String type;
    private final Map<String, String> metadata = new HashMap<>();

    public Builder account(String account) {
        this.account = account;
        return this;
    }

    public Builder amount(BigDecimal amount) {
        this.amount = amount;
        return this;
    }

    public Builder type(String type) {
        this.type = type;
        return this;
    }
```

```
        public Builder addMetadata(String key, String value) {
            metadata.put(key, value);
            return this;
        }

        public LedgerEntry build() {
            if (account == null || amount == null || type == null)
                throw new IllegalStateException("Required fields missing");
            return new LedgerEntry(this);
        }
    }

    public String getId() { return id; }
    public String getAccount() { return account; }
    public BigDecimal getAmount() { return amount; }
    public String getType() { return type; }
    public LocalDateTime getTimestamp() { return timestamp; }
    public Map<String, String> getMetadata() { return metadata; }

    @Override
    public String toString() {
        return "[%s] %s %s %s -> %s".formatted(timestamp, id, type, amount, account);
    }
}
```

```
// ---------------------------------------------
// Event Interface and Visitor for extensibility
// ---------------------------------------------
interface LedgerEvent {
    void accept(LedgerVisitor visitor);
}

class LedgerEntryEvent implements LedgerEvent {
    private final LedgerEntry entry;

    public LedgerEntryEvent(LedgerEntry entry) {
        this.entry = entry;
    }

    public LedgerEntry getEntry() {
        return entry;
    }

    @Override
    public void accept(LedgerVisitor visitor) {
        visitor.visit(this);
    }
}

interface LedgerVisitor {
    void visit(LedgerEntryEvent event);
}

// ---------------------------------------------
// Observer Pattern: Ledger Event Dispatcher
// ---------------------------------------------
interface LedgerObserver {
    void onLedgerEvent(LedgerEvent event);
}

class EventDispatcher {
    private final List<LedgerObserver> observers = new CopyOnWriteArrayList<>();

    public void register(LedgerObserver observer) {
        observers.add(observer);
    }

    public void dispatch(LedgerEvent event) {
        observers.parallelStream().forEach(o -> o.onLedgerEvent(event));
    }
}

// ---------------------------------------------
// Cache Layer with Thread-Safe Ledger Index
// ---------------------------------------------
```

```
class LedgerCache {
    private final ConcurrentMap<String, List<LedgerEntry>> accountMap = new
ConcurrentHashMap<>();


    public void addEntry(LedgerEntry entry) {
        accountMap.computeIfAbsent(entry.getAccount(), k -> new
CopyOnWriteArrayList<>()).add(entry);
    }


    public List<LedgerEntry> getEntries(String account) {
        return accountMap.getOrDefault(account, Collections.emptyList());
    }


    public BigDecimal getBalance(String account) {
        return getEntries(account).stream()
            .map(e -> e.getType().equals("credit") ? e.getAmount() : e.getAmount().negate())
            .reduce(BigDecimal.ZERO, BigDecimal::add);
    }
}


// ---------------------------------------------
// Simulated Ledger Database (Thread-safe Queue)
// ---------------------------------------------
class LedgerDatabase {
    private final BlockingQueue<LedgerEntry> journal = new LinkedBlockingQueue<>();
    }

    public void persist(LedgerEntry entry) {
        journal.offer(entry);
    }

    public List<LedgerEntry> getAllEntries() {
        return new ArrayList<>(journal);
    }
}

// ---------------------------------------------
// Main Service: Ledger Engine with DI
// ---------------------------------------------
class LedgerEngine implements LedgerVisitor, LedgerObserver {
    @Inject
    private LedgerDatabase db;

    @Inject
    private LedgerCache cache;

    @Override
    public void visit(LedgerEntryEvent event) {
        LedgerEntry entry = event.getEntry();
        db.persist(entry);
        cache.addEntry(entry);
    }
    @Override
    public void onLedgerEvent(LedgerEvent event) {
        event.accept(this);
    }

    public BigDecimal queryBalance(String account) {
        return cache.getBalance(account);
    }

    public List<LedgerEntry> history(String account) {
        return cache.getEntries(account);
    }
}

// ---------------------------------------------
// Main Application: Running QuantumLedger™
// ---------------------------------------------
public class QuantumLedger {
    public static void main(String[] args) throws Exception {
        EventDispatcher dispatcher = new EventDispatcher();
        LedgerEngine engine = SingletonRegistry.getInstance(LedgerEngine.class);
        dispatcher.register(engine);

        List<LedgerEntry> testEntries = List.of(
            new LedgerEntry.Builder().account("wallet_A").amount(BigDecimal.valueOf(1000)).type("credit").build(),
            new LedgerEntry.Builder().account("wallet_A").amount(BigDecimal.valueOf(200)).type("debit").build(),
            new LedgerEntry.Builder().account("wallet_B").amount(BigDecimal.valueOf(500)).type("credit").build()
        );

        testEntries.forEach(entry -> dispatcher.dispatch(new LedgerEntryEvent(entry)));

        // Display balance and history
        System.out.println("Balance (wallet_A): " + engine.queryBalance("wallet_A"));
        engine.history("wallet_A").forEach(System.out::println);

        System.out.println("Balance (wallet_B): " + engine.queryBalance("wallet_B"));
        engine.history("wallet_B").forEach(System.out::println);
    }
}
```

Key Concepts Highlighted

Concept	Purpose / HR Impression
Dependency Injection (DI)	Manual reflection-based DI shows deep knowledge without relying on Spring.
Builder Pattern	Clean immutability and object construction.

Concept	Purpose / HR Impression
Visitor & Observer Pattern	Complex extensibility and event-driven architecture.
Thread-Safe Data Structures	Used CopyOnWriteArrayList, ConcurrentMap, and BlockingQueue correctly.
Java Streams & Lambdas	Functional programming idioms for aggregation and filtering.
Custom Annotation + Reflection	Shows mastery over Java's meta-programming capabilities.
Real-world use case	Simulates high-performance financial ledger logic akin to fintech or blockchain tech.
Immutability & Concurrency	Perfect balance between safe data and multi-threaded performance.