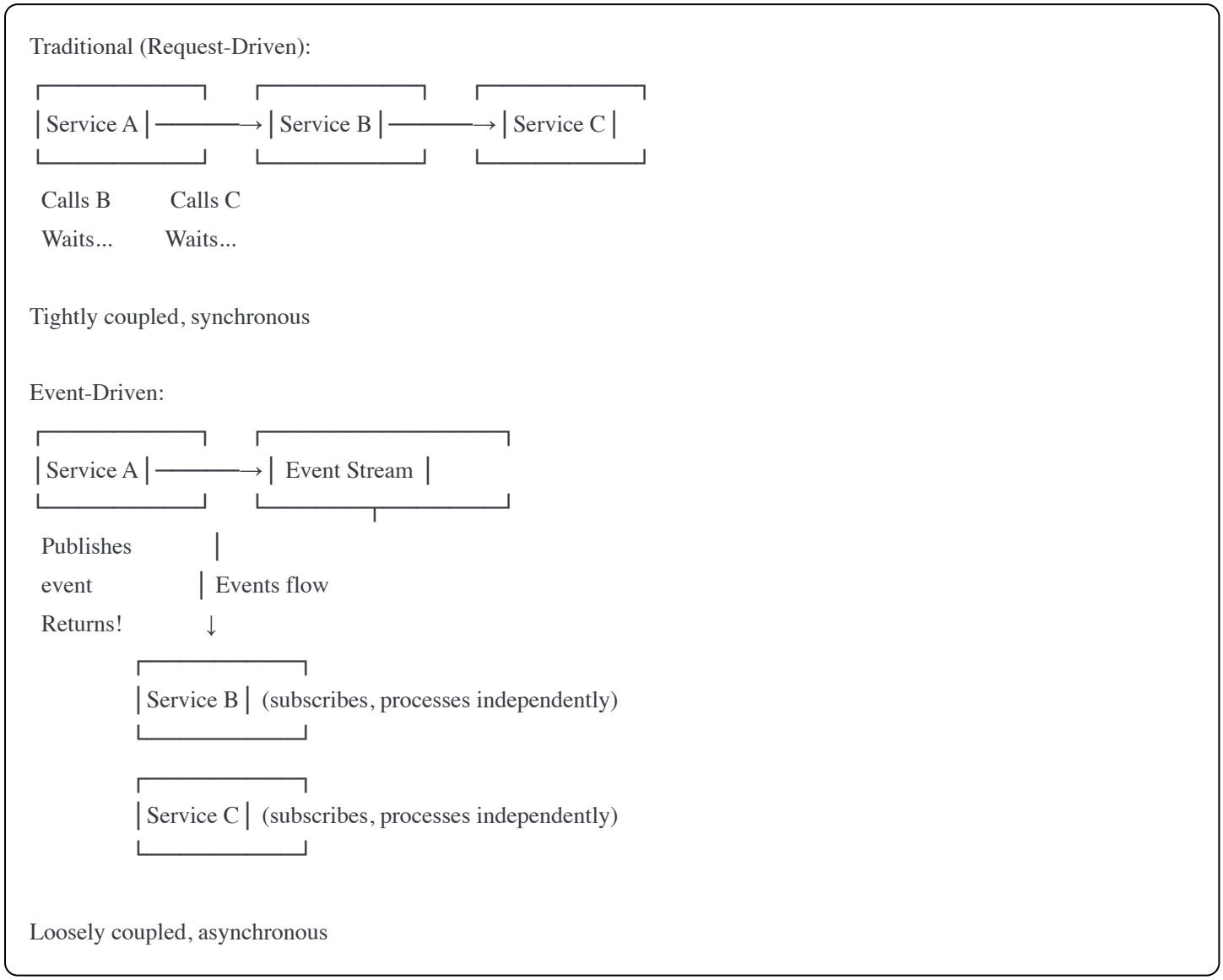
Chapter 11: Event-Driven Architecture

Introduction: What is Event-Driven Architecture?

Event-Driven Architecture (EDA): Systems communicate through events rather than direct calls.



---

1. Event Sourcing

What is Event Sourcing?

Traditional Approach (Store Current State):

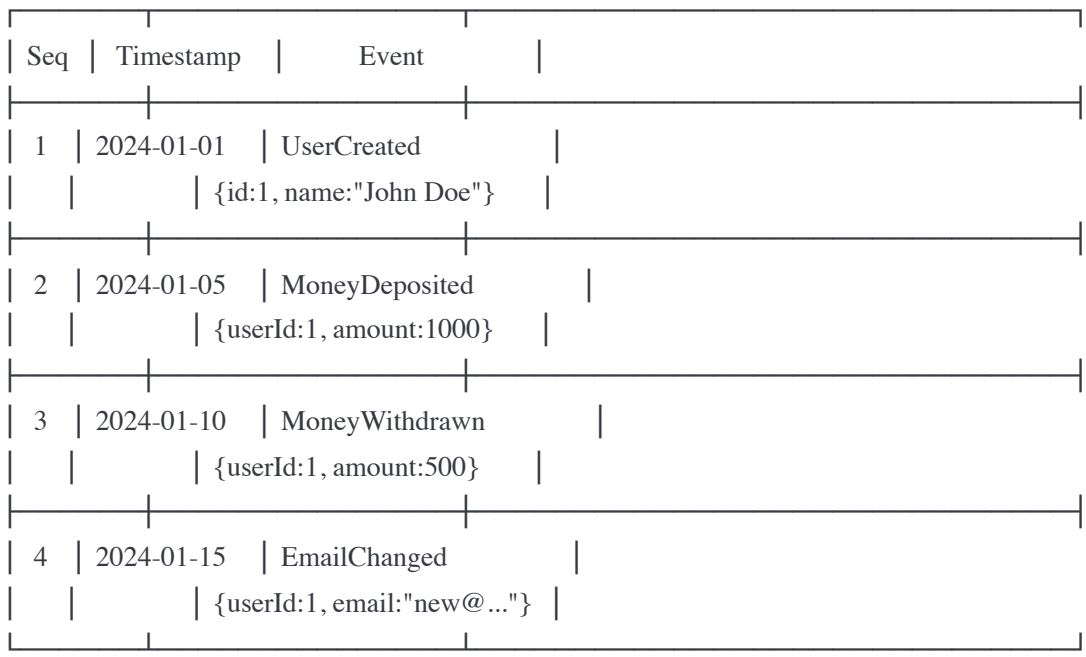
Database stores CURRENT state only:

Users Table:

Problem: Lost history!
- How did balance become $500?
- When did name change?
- Can't audit changes
- Can't replay history

Event Sourcing (Store All Changes):

Event Store (Immutable Log):



Current state = Replay all events
Balance = 0 + 1000 - 500 = $500 ✓

Benefits:
- Complete audit trail
- Can reconstruct any point in time
- Can replay events
- Never lose data

Event Sourcing Implementation

javascript

```
// Event types
const EventTypes = {
  ACCOUNT_CREATED: 'ACCOUNT_CREATED',
  MONEY_DEPOSITED: 'MONEY_DEPOSITED',
  MONEY_WITHDRAWN: 'MONEY_WITHDRAWN',
  ACCOUNT_CLOSED: 'ACCOUNT_CLOSED'
};

// Event Store
class EventStore {
  constructor() {
    this.events = [];  // In production: use database
  }

  append(event) {
    event.sequence = this.events.length + 1;
    event.timestamp = new Date().toISOString();
    this.events.push(event);
    console.log('Event stored:', event);
  }

  getEvents(entityId) {
    return this.events.filter(e => e.entityId === entityId);
  }

  getAllEvents() {
    return this.events;
  }
}

// Aggregate (Entity rebuilt from events)
class BankAccount {
  constructor(accountId) {
    this.accountId = accountId;
    this.balance = 0;
    this.isOpen = false;
    this.owner = null;
    this.version = 0;
  }

  // Rebuild state by replaying events
  static fromEvents(events) {
    const account = new BankAccount(events[0].entityId);

    events.forEach(event => {
      account.apply(event);
```

```
    });

    return account;
  }

  // Apply event to update state
  apply(event) {
    switch (event.type) {
      case EventTypes.ACCOUNT_CREATED:
        this.owner = event.data.owner;
        this.isOpen = true;
        break;

      case EventTypes.MONEY_DEPOSITED:
        this.balance += event.data.amount;
        break;

      case EventTypes.MONEY_WITHDRAWN:
        this.balance -= event.data.amount;
        break;

      case EventTypes.ACCOUNT_CLOSED:
        this.isOpen = false;
        break;
    }

    this.version++;
  }

  getState() {
    return {
      accountId: this.accountId,
      owner: this.owner,
      balance: this.balance,
      isOpen: this.isOpen,
      version: this.version
    };
  }
}

// Command handlers (create events)
class BankAccountService {
  constructor(eventStore) {
    this.eventStore = eventStore;
  }

  createAccount(accountId, owner) {
```

```
  const event = {
    type: EventTypes.ACCOUNT_CREATED,
    entityId: accountId,
    data: { owner }
  };

  this.eventStore.append(event);
  return event;
}

deposit(accountId, amount) {
  // Load account from events
  const events = this.eventStore.getEvents(accountId);
  const account = BankAccount.fromEvents(events);

  // Validate
  if (!account.isOpen) {
    throw new Error('Account is closed');
  }

  // Create event
  const event = {
    type: EventTypes.MONEY_DEPOSITED,
    entityId: accountId,
    data: { amount }
  };

  this.eventStore.append(event);
  return event;
}

withdraw(accountId, amount) {
  // Load account from events
  const events = this.eventStore.getEvents(accountId);
  const account = BankAccount.fromEvents(events);

  // Validate
  if (!account.isOpen) {
    throw new Error('Account is closed');
  }

  if (account.balance < amount) {
    throw new Error('Insufficient funds');
  }

  // Create event
  const event = {
```

```
      type: EventTypes.MONEY_WITHDRAWN,
      entityId: accountId,
      data: { amount }
    };

    this.eventStore.append(event);
    return event;
  }

  getAccount(accountId) {
    const events = this.eventStore.getEvents(accountId);

    if (events.length === 0) {
      return null;
    }

    return BankAccount.fromEvents(events);
  }
}

// Usage
const eventStore = new EventStore();
const service = new BankAccountService(eventStore);

// Create account
service.createAccount('acc-123', 'John Doe');

// Deposit money
service.deposit('acc-123', 1000);
service.deposit('acc-123', 500);

// Withdraw money
service.withdraw('acc-123', 300);

// Get current state (by replaying events)
const account = service.getAccount('acc-123');
console.log('Current state:', account.getState());

// Output:
// Event stored: {type: ACCOUNT_CREATED, entityId: acc-123, ...}
// Event stored: {type: MONEY_DEPOSITED, entityId: acc-123, amount: 1000}
// Event stored: {type: MONEY_DEPOSITED, entityId: acc-123, amount: 500}
// Event stored: {type: MONEY_WITHDRAWN, entityId: acc-123, amount: 300}
// Current state: {
//   accountId: 'acc-123',
//   owner: 'John Doe',
//   balance: 1200,  // 0 + 1000 + 500 - 300
```

```
// isOpen: true,
// version: 4
// }
```

Event Sourcing Benefits

1. COMPLETE AUDIT TRAIL
   Know exactly what happened and when
   Example: Banking, healthcare, legal systems

2. TIME TRAVEL
   Reconstruct state at any point in time

   Timeline:
   10:00 - Balance: $0
   10:05 - Deposit $1000 → Balance: $1000
   10:10 - Withdraw $500 → Balance: $500

   Query: What was balance at 10:07?
   Replay events up to 10:07 → $1000!

3. EVENT REPLAY
   Fix bugs by replaying events with corrected logic

   Example:
   Bug: Interest calculation wrong
   Fix: Update interest calculation
   Replay: Recalculate from all events
   Result: Corrected balances!

4. DEBUGGING
   Reproduce issues by replaying same events

5. ANALYTICS
   Build new reports from historical events
   Don't need to have thought of report beforehand!

6. EVENTUAL CONSISTENCY
   Different views can be out of sync temporarily
   All eventually consistent with event log

Event Sourcing Challenges

1. QUERY COMPLEXITY
   Problem: Hard to query "Which users have balance > $1000?"
   Must replay ALL events to get current state

   Solution: CQRS (next section!)

2. STORAGE GROWTH
   Events accumulate forever

   Solutions:
   - Snapshots (store state periodically)
   - Event archival (move old events to cold storage)
   - Event deletion (for GDPR compliance)

3. SCHEMA EVOLUTION
   Event schema changes over time

   Solution: Version events

   Event V1: { type: 'MoneyDeposited', amount: 100 }
   Event V2: { type: 'MoneyDeposited', amount: 100, currency: 'USD' }

   Handle both versions when replaying

4. COMPLEXITY
   Harder to understand than CRUD
   Steeper learning curve

Snapshots (Performance Optimization)

Without Snapshots:
To get current state, replay ALL events:
Account has 10,000 events
Replay 10,000 events = 500ms (slow!)

With Snapshots:
Every 1,000 events, save snapshot:

Events: 1-1000 → Snapshot 1 (version 1000)
Events: 1001-2000 → Snapshot 2 (version 2000)
...
Events: 9001-10000 → Current

To get current state:

1. Load snapshot at version 9000 (fast!)
2. Replay events 9001-10000 (only 1000 events)

Total: 50ms (10x faster!)

Implementation:

```javascript
```

```
class EventStoreWithSnapshots {
  constructor() {
    this.events = [];
    this.snapshots = new Map();  // version -> snapshot
    this.snapshotInterval = 100;  // Snapshot every 100 events
  }

  append(event) {
    event.sequence = this.events.length + 1;
    event.timestamp = new Date().toISOString();
    this.events.push(event);

    // Create snapshot if needed
    if (event.sequence % this.snapshotInterval === 0) {
      this.createSnapshot(event.entityId);
    }
  }

  createSnapshot(entityId) {
    const events = this.getEvents(entityId);
    const account = BankAccount.fromEvents(events);

    this.snapshots.set(`${entityId}:${account.version}`, {
      entityId,
      version: account.version,
      state: account.getState(),
      timestamp: new Date().toISOString()
    });

    console.log(`Snapshot created for ${entityId} at version ${account.version}`);
  }

  getEvents(entityId) {
    return this.events.filter(e => e.entityId === entityId);
  }

  getAccountFast(accountId) {
    const events = this.getEvents(accountId);

    if (events.length === 0) {
      return null;
    }

    // Find latest snapshot
    let latestSnapshot = null;
    let latestVersion = 0;
```

```
    for (const [key, snapshot] of this.snapshots) {
      if (snapshot.entityId === accountId && snapshot.version > latestVersion) {
        latestSnapshot = snapshot;
        latestVersion = snapshot.version;
      }
    }

    if (latestSnapshot) {
      console.log(`Using snapshot at version ${latestVersion}`);

      // Create account from snapshot
      const account = new BankAccount(accountId);
      Object.assign(account, latestSnapshot.state);

      // Replay only events after snapshot
      const eventsAfterSnapshot = events.filter(e => e.sequence > latestVersion);
      eventsAfterSnapshot.forEach(e => account.apply(e));

      console.log(`Replayed ${eventsAfterSnapshot.length} events (vs ${events.length} total)`);

      return account;
    }

    // No snapshot, replay all events
    console.log(`No snapshot, replaying all ${events.length} events`);
    return BankAccount.fromEvents(events);
  }
}

// Usage
const eventStore = new EventStoreWithSnapshots();
const service = new BankAccountService(eventStore);

// Create account
service.createAccount('acc-123', 'John Doe');

// Make 250 transactions
for (let i = 0; i < 250; i++) {
  if (i % 2 === 0) {
    service.deposit('acc-123', 100);
  } else {
    service.withdraw('acc-123', 50);
  }
}

// Get account (uses snapshot!)
```

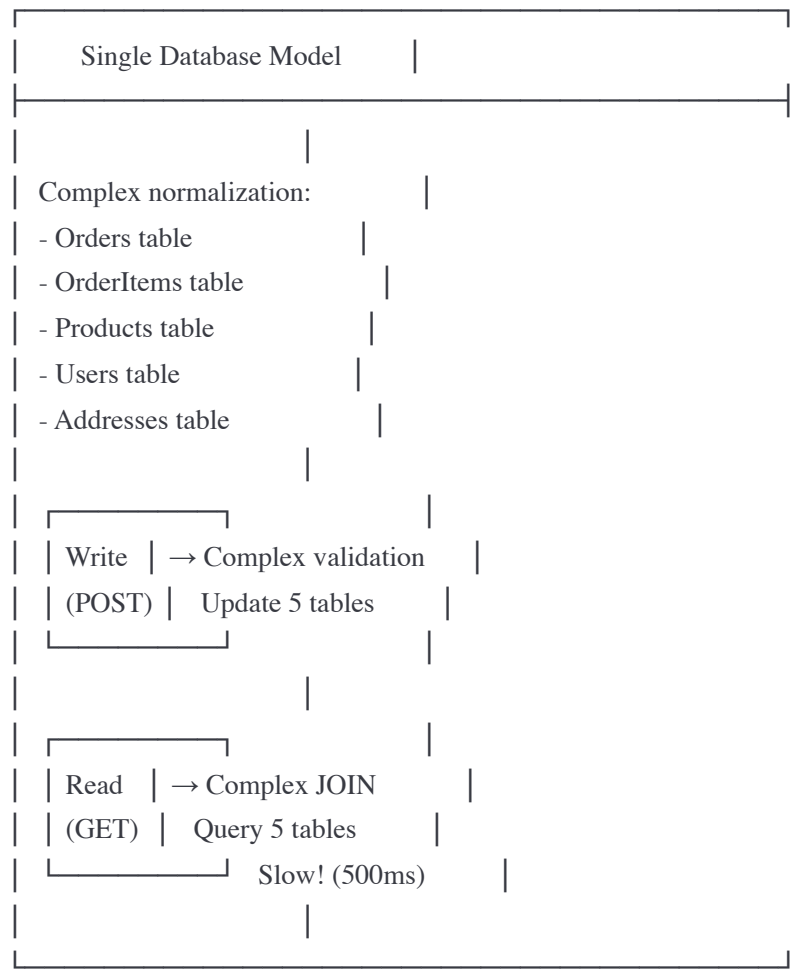
```
const account = eventStore.getAccountFast('acc-123');

// Output:
// Snapshot created for acc-123 at version 100
// Snapshot created for acc-123 at version 200
// Using snapshot at version 200
// Replayed 51 events (vs 251 total)
```

---

2. CQRS (Command Query Responsibility Segregation)

The Problem

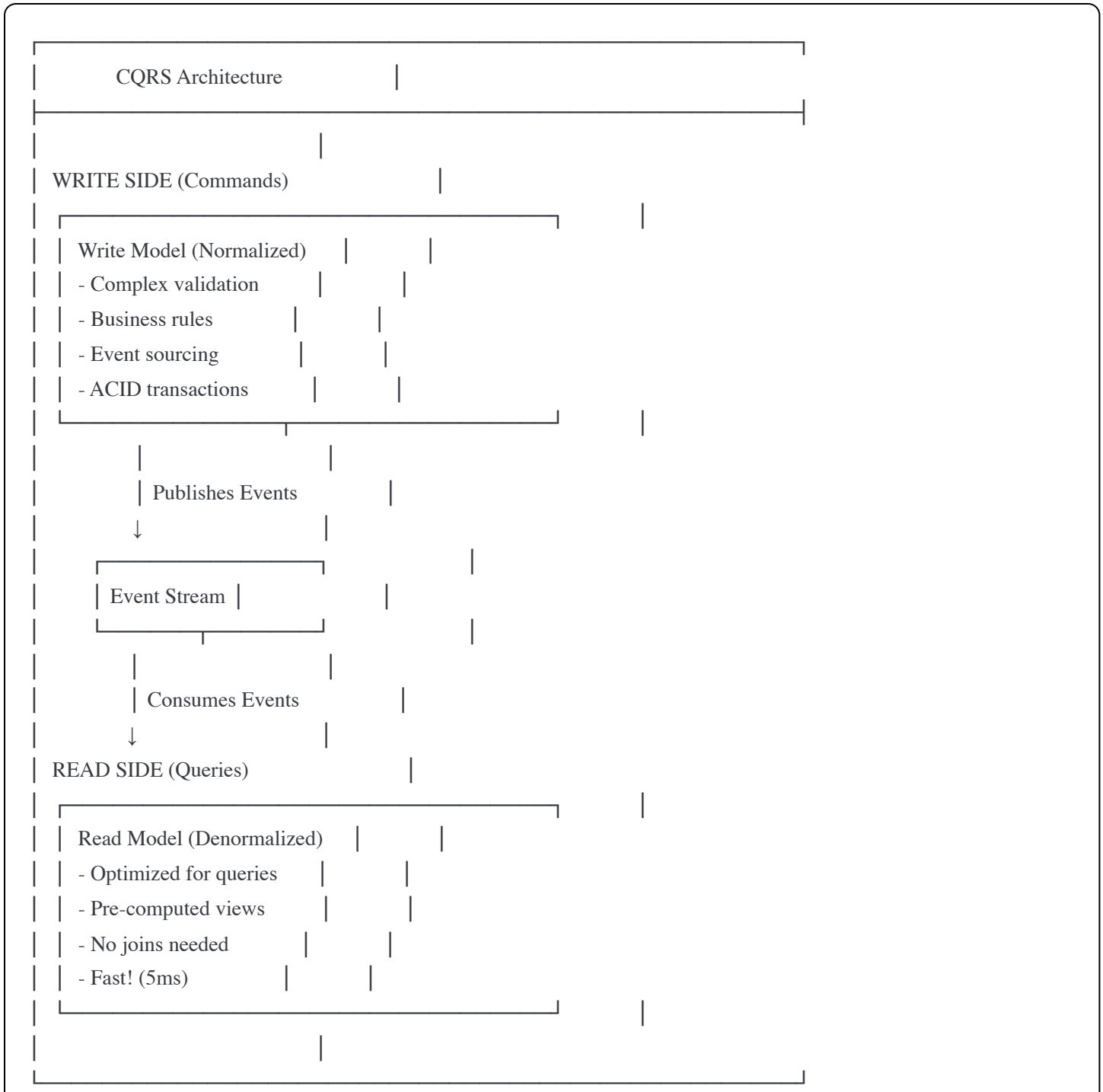
Traditional CRUD (same model for read and write):



Problem: Reads and writes have different needs!

CQRS Solution

Separate models for read and write:



Write: Optimized for data integrity

Read: Optimized for performance

CQRS Implementation

javascript

```
// WRITE SIDE
class CommandHandler {
  constructor(eventStore) {
    this.eventStore = eventStore;
  }

  // Commands (write operations)
  async createOrder(command) {
    const { orderId, userId, items } = command;

    // Validate command
    if (!items || items.length === 0) {
      throw new Error('Order must have items');
    }

    // Check business rules
    const total = items.reduce((sum, item) => sum + item.price * item.quantity, 0);

    // Create event
    const event = {
      type: 'ORDER_CREATED',
      entityId: orderId,
      data: {
        orderId,
        userId,
        items,
        total,
        status: 'pending'
      }
    };

    // Store event
    this.eventStore.append(event);

    return event;
  }

  async addItemToOrder(command) {
    const { orderId, item } = command;

    // Load order from events
    const events = this.eventStore.getEvents(orderId);
    const order = Order.fromEvents(events);

    // Validate
    if (order.status !== 'pending') {
```

```
      throw new Error('Cannot modify confirmed order');
    }

    // Create event
    const event = {
      type: 'ITEM_ADDED',
      entityId: orderId,
      data: { item }
    };

    this.eventStore.append(event);
    return event;
  }

  async confirmOrder(command) {
    const { orderId } = command;

    const event = {
      type: 'ORDER_CONFIRMED',
      entityId: orderId,
      data: { confirmedAt: new Date().toISOString() }
    };

    this.eventStore.append(event);
    return event;
  }
}

// READ SIDE
class ReadModelUpdater {
  constructor(eventStore, readDatabase) {
    this.eventStore = eventStore;
    this.readDb = readDatabase;  // Denormalized database

    // Subscribe to events
    this.eventStore.on('event', (event) => this.handleEvent(event));
  }

  async handleEvent(event) {
    switch (event.type) {
      case 'ORDER_CREATED':
        await this.updateOrderView(event);
        await this.updateUserOrdersView(event);
        await this.updateSalesStatistics(event);
        break;

      case 'ITEM_ADDED':
```

```
      await this.updateOrderView(event);
      break;

    case 'ORDER_CONFIRMED':
      await this.updateOrderView(event);
      await this.updateInventoryView(event);
      break;
  }
}

async updateOrderView(event) {
  // Denormalized view: All order data in one document
  const order = {
    orderId: event.entityId,
    userId: event.data.userId,
    items: event.data.items,
    total: event.data.total,
    status: event.data.status,
    createdAt: event.timestamp
  };

  // Store in read-optimized database (e.g., MongoDB)
  await this.readDb.collection('orders_view').updateOne(
    { orderId: event.entityId },
    { $set: order },
    { upsert: true }
  );

  console.log('Order view updated:', event.entityId);
}

async updateUserOrdersView(event) {
  // Denormalized view: User with all their orders
  await this.readDb.collection('users_orders_view').updateOne(
    { userId: event.data.userId },
    {
      $push: {
        orders: {
          orderId: event.entityId,
          total: event.data.total,
          createdAt: event.timestamp
        }
      }
    },
    { upsert: true }
  );
```

```
    console.log('User orders view updated');
  }

  async updateSalesStatistics(event) {
    // Pre-computed statistics
    const date = event.timestamp.split('T')[0]; // YYYY-MM-DD

    await this.readDb.collection('daily_sales').updateOne(
      { date },
      {
        $inc: {
          orderCount: 1,
          totalRevenue: event.data.total
        }
      },
      { upsert: true }
    );

    console.log('Sales statistics updated');
  }

  async updateInventoryView(event) {
    // Update inventory when order confirmed
    for (const item of event.data.items) {
      await this.readDb.collection('inventory_view').updateOne(
        { productId: item.productId },
        { $inc: { available: -item.quantity } }
      );
    }
  }
}

// QUERY SIDE
class QueryHandler {
  constructor(readDatabase) {
    this.readDb = readDatabase;
  }

  // Queries (read operations) - Super fast!
  async getOrder(orderId) {
    // Single document lookup, no joins!
    return await this.readDb.collection('orders_view').findOne({ orderId });
  }

  async getUserOrders(userId) {
    // Denormalized, all data in one document
    return await this.readDb.collection('users_orders_view').findOne({ userId });
```

```
  }

  async getDailySales(date) {
    // Pre-computed statistics
    return await this.readDb.collection('daily_sales').findOne({ date });
  }

  async searchOrders(criteria) {
    // Can have multiple specialized indexes
    return await this.readDb.collection('orders_view').find(criteria).toArray();
  }
}

// Usage
const commandHandler = new CommandHandler(eventStore);
const queryHandler = new QueryHandler(readDatabase);

// Write (Command)
await commandHandler.createOrder({
  orderId: 'order-123',
  userId: 'user-456',
  items: [
    { productId: 'prod-1', quantity: 2, price: 50 }
  ]
});

// Read (Query) - Fast! No event replay needed
const order = await queryHandler.getOrder('order-123');
const userOrders = await queryHandler.getUserOrders('user-456');
const stats = await queryHandler.getDailySales('2024-01-20');
```

CQRS Benefits and Challenges

```
✓ ADVANTAGES:
  - Read model optimized for queries (10-100x faster)
  - Write model optimized for business logic
  - Can have multiple read models (different views)
  - Scale reads and writes independently
  - No complex JOINs on read side

✗ DISADVANTAGES:
  - Eventual consistency (read may be slightly behind write)
  - More complex architecture
  - Two databases to maintain
  - Synchronization between models
```

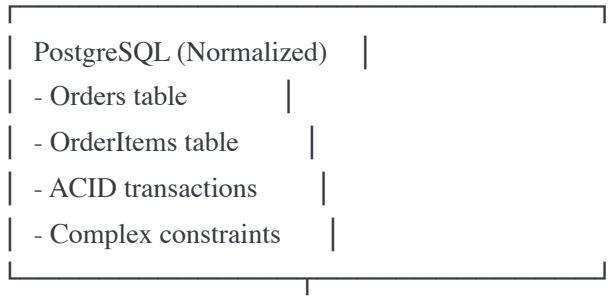
When to use CQRS:
✓ Complex domain logic
✓ High read:write ratio
✓ Need multiple views of same data
✓ Performance critical reads
✓ Can tolerate eventual consistency

When NOT to use:
✗ Simple CRUD applications
✗ Need strong consistency
✗ Small team (complexity overhead)

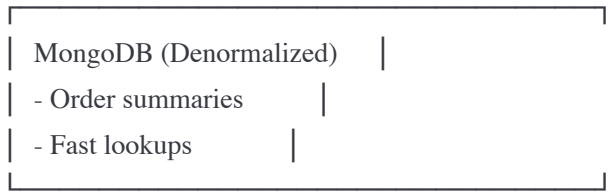
CQRS with Different Databases

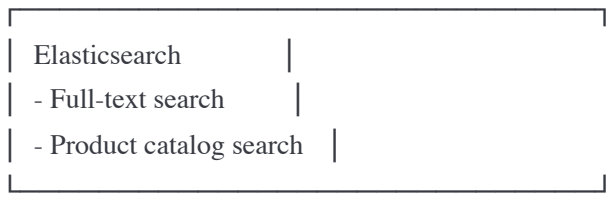
Example: E-commerce Platform

WRITE SIDE:



READ SIDE (Multiple Views):





Each optimized for its purpose!

---

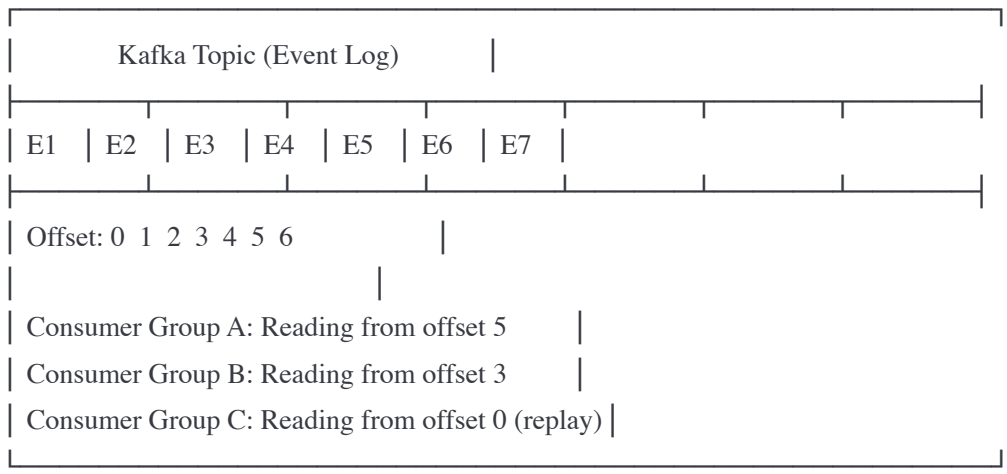
3. Event Streaming Platforms

Kafka as Event Streaming Platform

Beyond Simple Messaging:

Traditional Message Queue:
Producer → Queue → Consumer → Message deleted

Kafka Event Stream:
Producer → Topic (Append-only log) → Multiple consumers
→ Messages retained
→ Can replay anytime



Key feature: Log is retained (configurable)
Can replay from any offset!

Event Streaming Use Cases

1. EVENT SOURCING
   Store all state changes as events
   Rebuild state by replaying

2. CHANGE DATA CAPTURE (CDC)
   Capture database changes as events
   Replicate to other systems

3. REAL-TIME ANALYTICS
   Process events as they arrive

Build dashboards, metrics

4. MICROSERVICES COMMUNICATION
Services publish/subscribe to events
Loose coupling

5. AUDIT LOG
Immutable record of all actions
Compliance, debugging

6. STREAM PROCESSING
Transform, filter, aggregate events
Build derived streams

Kafka Streams (Processing Events)

```javascript
```

```
// Kafka Streams allows processing events in real-time

const { Kafka } = require('kafkajs');

const kafka = new Kafka({
  clientId: 'stream-processor',
  brokers: ['localhost:9092']
});

// Simple stream processing
async function processEventStream() {
  const consumer = kafka.consumer({ groupId: 'analytics-processor' });
  const producer = kafka.producer();

  await consumer.connect();
  await producer.connect();

  await consumer.subscribe({ topic: 'user-events', fromBeginning: true });

  // Aggregate by user ID
  const userStats = new Map();

  await consumer.run({
    eachMessage: async ({ topic, partition, message }) => {
      const event = JSON.parse(message.value.toString());

      // Process event
      const userId = event.userId;

      if (!userStats.has(userId)) {
        userStats.set(userId, {
          userId,
          eventCount: 0,
          lastSeen: null,
          actions: []
        });
      }

      const stats = userStats.get(userId);
      stats.eventCount++;
      stats.lastSeen = event.timestamp;
      stats.actions.push(event.type);

      // Publish processed event to output topic
      if (stats.eventCount % 10 === 0) {
        await producer.send({
```

```
        topic: 'user-stats',
        messages: [{
          key: userId,
          value: JSON.stringify(stats)
        }]
      });

      console.log(`Published stats for user ${userId}`);
    }
  }
});
}

// Windowed aggregation (count events per 5-minute window)
async function windowedAggregation() {
  const consumer = kafka.consumer({ groupId: 'windowed-analytics' });
  const producer = kafka.producer();

  await consumer.connect();
  await producer.connect();
  await consumer.subscribe({ topic: 'page-views', fromBeginning: true });

  const windowSize = 5 * 60 * 1000;  // 5 minutes
  const windows = new Map();  // windowKey -> count

  await consumer.run({
    eachMessage: async ({ message }) => {
      const event = JSON.parse(message.value.toString());
      const timestamp = new Date(event.timestamp).getTime();

      // Determine window
      const windowStart = Math.floor(timestamp / windowSize) * windowSize;
      const windowKey = `${event.pageId}:${windowStart}`;

      // Increment count for this window
      const currentCount = windows.get(windowKey) || 0;
      windows.set(windowKey, currentCount + 1);

      // Publish window results
      if ((currentCount + 1) % 100 === 0) {
        await producer.send({
          topic: 'page-views-windowed',
          messages: [{
            key: event.pageId,
            value: JSON.stringify({
              pageId: event.pageId,
              windowStart: new Date(windowStart).toISOString(),
```

```
            count: currentCount + 1
          })
        }]
      });
    }

    // Cleanup old windows
    const now = Date.now();
    for (const [key, _] of windows) {
      const [_, winStart] = key.split(':');
      if (now - parseInt(winStart) > 60 * 60 * 1000) {  // 1 hour old
        windows.delete(key);
      }
    }
  }
 });
}

// Join two streams
async function joinStreams() {
  const consumer = kafka.consumer({ groupId: 'stream-joiner' });
  const producer = kafka.producer();

  await consumer.connect();
  await producer.connect();

  // Subscribe to multiple topics
  await consumer.subscribe({ topics: ['orders', 'payments'], fromBeginning: true });

  // Store state for joining
  const orderState = new Map();
  const paymentState = new Map();

  await consumer.run({
    eachMessage: async ({ topic, message }) => {
      const event = JSON.parse(message.value.toString());

      if (topic === 'orders') {
        orderState.set(event.orderId, event);

        // Check if payment exists
        if (paymentState.has(event.orderId)) {
          const payment = paymentState.get(event.orderId);
          await publishJoined(event, payment);
        }

      } else if (topic === 'payments') {
```

```
      paymentState.set(event.orderId, event);

      // Check if order exists
      if (orderState.has(event.orderId)) {
        const order = orderState.get(event.orderId);
        await publishJoined(order, event);
      }
    }
  }
});

async function publishJoined(order, payment) {
  // Publish joined event
  await producer.send({
    topic: 'order-payment-joined',
    messages: [{
      key: order.orderId,
      value: JSON.stringify({
        orderId: order.orderId,
        orderTotal: order.total,
        paymentAmount: payment.amount,
        paymentStatus: payment.status
      })
    }]
  });
 }
}
```

---

4. Real-Time Data Processing

Stream Processing Patterns

1. Stateless Transformation

javascript

```javascript
// Filter, map, transform events

async function filterHighValueOrders() {
  const consumer = kafka.consumer({ groupId: 'high-value-filter' });
  const producer = kafka.producer();

  await consumer.connect();
  await producer.connect();
  await consumer.subscribe({ topic: 'all-orders', fromBeginning: true });

  await consumer.run({
    eachMessage: async ({ message }) => {
      const order = JSON.parse(message.value.toString());

      // Filter: Only high-value orders
      if (order.total > 1000) {
        // Transform: Add classification
        const enriched = {
          ...order,
          classification: 'high-value',
          priority: 'urgent'
        };

        // Publish to new topic
        await producer.send({
          topic: 'high-value-orders',
          messages: [{
            key: order.orderId,
            value: JSON.stringify(enriched)
          }]
        });

        console.log('High-value order detected:', order.orderId);
      }
    }
  });
}
```

---

2. Stateful Aggregation

javascript

```
// Count, sum, average over time

class StatefulAggregator {
  constructor() {
    this.state = new Map();  // userId -> aggregatedData
  }

  async processEvents() {
    const consumer = kafka.consumer({ groupId: 'user-aggregator' });
    const producer = kafka.producer();

    await consumer.connect();
    await producer.connect();
    await consumer.subscribe({ topic: 'user-actions', fromBeginning: true });

    await consumer.run({
      eachMessage: async ({ message }) => {
        const event = JSON.parse(message.value.toString());
        const userId = event.userId;

        // Get or initialize state
        if (!this.state.has(userId)) {
          this.state.set(userId, {
            userId,
            totalActions: 0,
            actionTypes: {},
            totalValue: 0,
            firstSeen: event.timestamp,
            lastSeen: null
          });
        }

        const userState = this.state.get(userId);

        // Update aggregated state
        userState.totalActions++;
        userState.actionTypes[event.type] = (userState.actionTypes[event.type] || 0) + 1;
        userState.totalValue += event.value || 0;
        userState.lastSeen = event.timestamp;

        // Publish aggregated state periodically
        if (userState.totalActions % 10 === 0) {
          await producer.send({
            topic: 'user-aggregates',
            messages: [{
              key: userId,
```

```
            value: JSON.stringify(userState)
          }]
        });
      }
    }
  });
 }
}

const aggregator = new StatefulAggregator();
aggregator.processEvents();
```

---

3. Complex Event Processing (CEP)

javascript

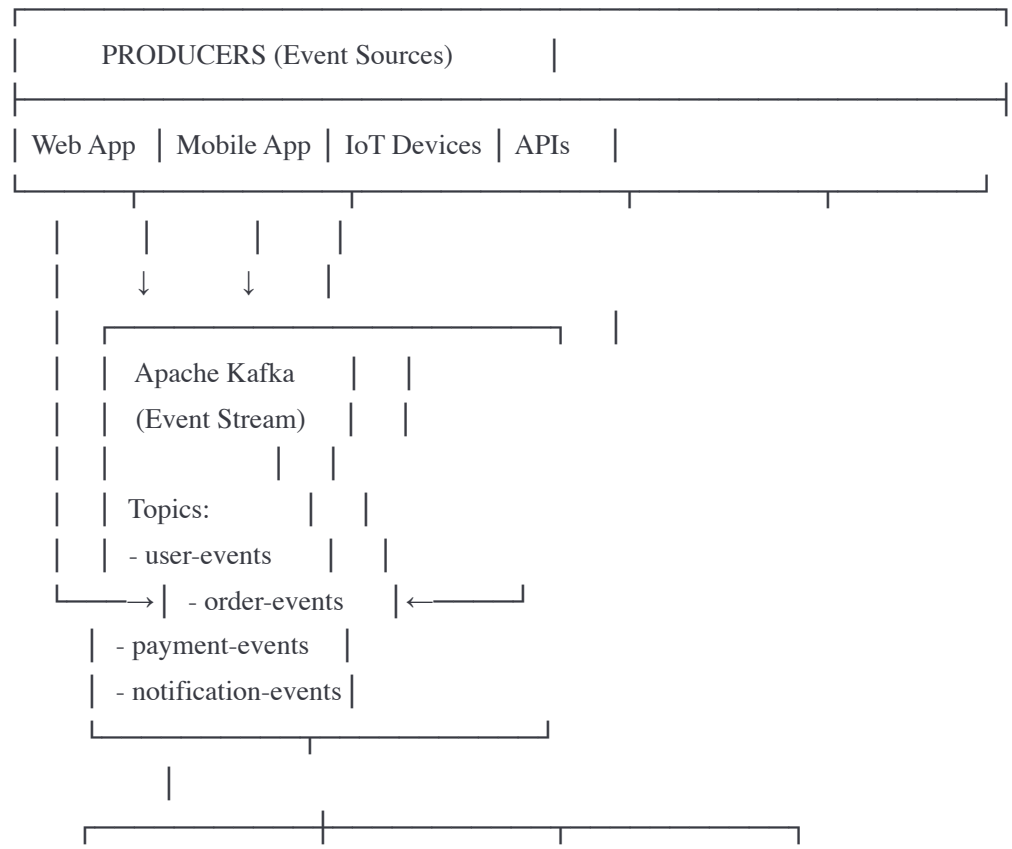
```
// Detect patterns in event streams

class FraudDetector {
  constructor() {
    this.userTransactions = new Map();
  }

  async detectFraud() {
    const consumer = kafka.consumer({ groupId: 'fraud-detector' });
    const producer = kafka.producer();

    await consumer.connect();
    await producer.connect();
    await consumer.subscribe({ topic: 'transactions', fromBeginning: false });

    await consumer.run({
      eachMessage: async ({ message }) => {
        const transaction = JSON.parse(message.value.toString());
        const userId = transaction.userId;

        // Track user's recent transactions
        if (!this.userTransactions.has(userId)) {
          this.userTransactions.set(userId, []);
        }

        const history = this.userTransactions.get(userId);
        history.push(transaction);

        // Keep last 10 transactions
        if (history.length > 10) {
          history.shift();
        }

        // Fraud detection patterns
        const fraudScore = this.calculateFraudScore(history, transaction);

        if (fraudScore > 0.8) {
          // Potential fraud detected!
          await producer.send({
            topic: 'fraud-alerts',
            messages: [{
              key: userId,
              value: JSON.stringify({
                userId,
                transactionId: transaction.id,
                fraudScore,
```

```
            reason: this.getFraudReason(history, transaction),
            timestamp: new Date().toISOString()
          })
        }]
      });

      console.log(`🚨 Fraud alert for user ${userId}: score ${fraudScore}`);
    }
  }
 });
}

calculateFraudScore(history, current) {
 let score = 0;

 // Pattern 1: Multiple large transactions in short time
 const recentLarge = history.filter(t =>
  t.amount > 1000 &&
  Date.now() - new Date(t.timestamp).getTime() < 300000  // 5 minutes
 );
 if (recentLarge.length >= 3) {
  score += 0.4;
 }

 // Pattern 2: Geographic anomaly
 if (history.length > 0) {
  const lastLocation = history[history.length - 1].location;
  if (this.distanceBetween(lastLocation, current.location) > 1000) {  // 1000 km
   const timeDiff = new Date(current.timestamp) - new Date(history[history.length - 1].timestamp);
   if (timeDiff < 3600000) {  // 1 hour
    score += 0.5;  // Impossible to travel 1000km in 1 hour
   }
  }
 }

 // Pattern 3: Unusual amount
 if (history.length > 5) {
  const avgAmount = history.reduce((sum, t) => sum + t.amount, 0) / history.length;
  if (current.amount > avgAmount * 5) {
   score += 0.3;
  }
 }

 return Math.min(score, 1.0);
}

getFraudReason(history, current) {
```

```
    const reasons = [];

    // Check patterns and add reasons
    const recentLarge = history.filter(t => t.amount > 1000);
    if (recentLarge.length >= 3) {
      reasons.push('Multiple large transactions');
    }

    // Add more reason detection...

    return reasons;
  }

  distanceBetween(loc1, loc2) {
    // Calculate distance (simplified)
    return Math.abs(loc1.lat - loc2.lat) + Math.abs(loc1.lng - loc2.lng) * 100;
  }
}

const detector = new FraudDetector();
detector.detectFraud();
```

Event Streaming Architecture

Complete Event-Driven System:



Each consumer:
- Processes at own pace
- Maintains own offset
- Can replay events
- Scales independently

Real-Time Processing Pipeline

javascript

```
// Complete real-time analytics pipeline

class RealTimeAnalyticsPipeline {
  constructor() {
    this.kafka = new Kafka({
      clientId: 'analytics-pipeline',
      brokers: ['localhost:9092']
    });
  }

  async start() {
    // Stage 1: Collect raw events
    await this.collectEvents();

    // Stage 2: Filter and enrich
    await this.filterAndEnrich();

    // Stage 3: Aggregate
    await this.aggregate();

    // Stage 4: Alert on anomalies
    await this.detectAnomalies();
  }

  async collectEvents() {
    const producer = this.kafka.producer();
    await producer.connect();

    // Simulate events
    setInterval(async () => {
      const event = {
        eventType: 'page_view',
        userId: `user-${Math.floor(Math.random() * 1000)}`,
        pageId: `page-${Math.floor(Math.random() * 100)}`,
        timestamp: new Date().toISOString(),
        duration: Math.floor(Math.random() * 60000)  // ms
      };

      await producer.send({
        topic: 'raw-events',
        messages: [{ value: JSON.stringify(event) }]
      });
    }, 100);  // 10 events/second
  }

  async filterAndEnrich() {
```

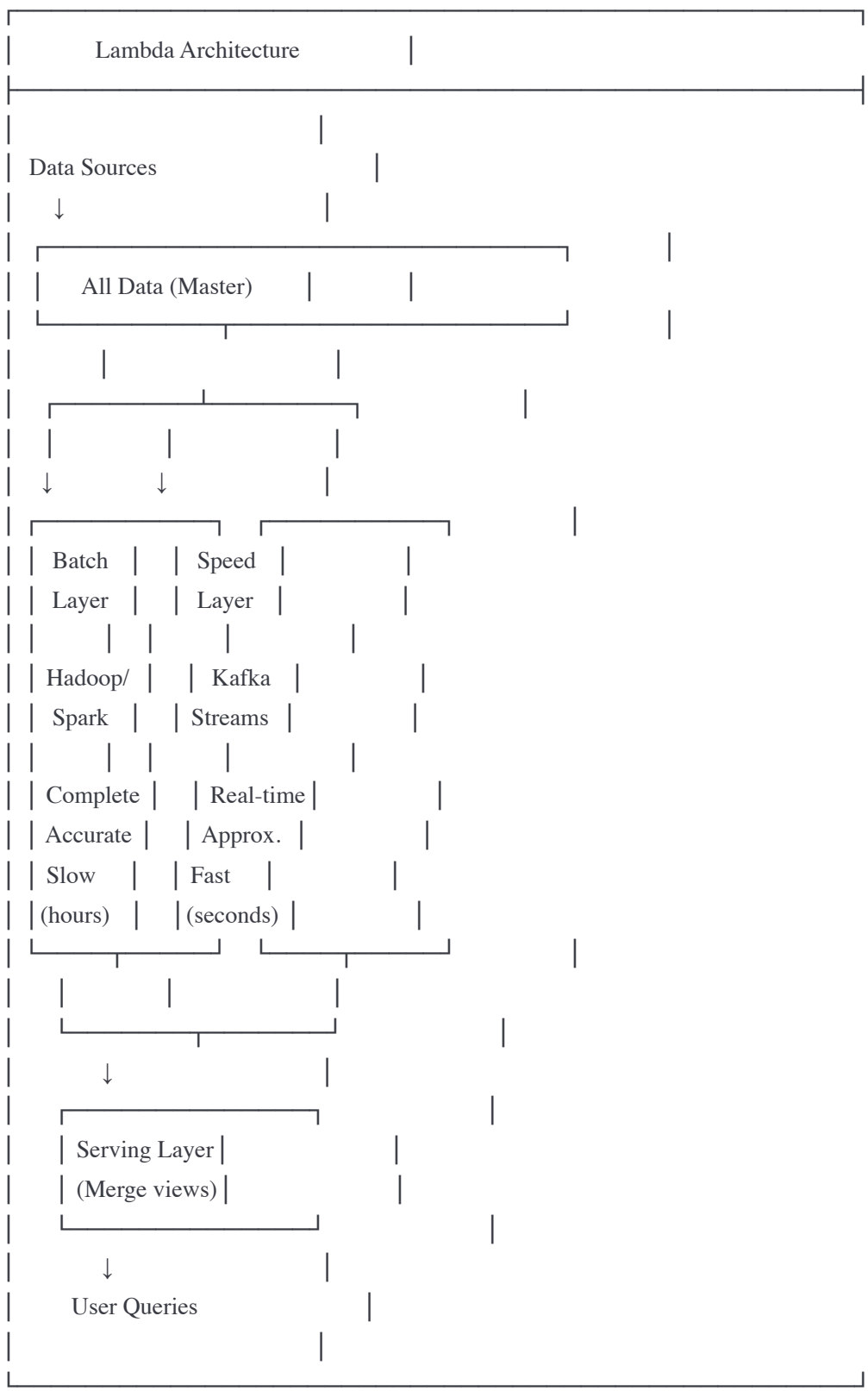
```
  const consumer = this.kafka.consumer({ groupId: 'filter-enrich' });
  const producer = this.kafka.producer();

  await consumer.connect();
  await producer.connect();
  await consumer.subscribe({ topic: 'raw-events', fromBeginning: false });

  await consumer.run({
    eachMessage: async ({ message }) => {
      const event = JSON.parse(message.value.toString());

      // Filter: Only page views > 10 seconds
      if (event.duration > 10000) {
        // Enrich: Add page info
        const pageInfo = await this.getPageInfo(event.pageId);

        const enriched = {
          ...event,
          pageTitle: pageInfo.title,
          pageCategory: pageInfo.category
        };

        // Publish to filtered topic
        await producer.send({
          topic: 'filtered-events',
          messages: [{ value: JSON.stringify(enriched) }]
        });
      }
    }
  });
}

async aggregate() {
  const consumer = this.kafka.consumer({ groupId: 'aggregator' });
  const producer = this.kafka.producer();

  await consumer.connect();
  await producer.connect();
  await consumer.subscribe({ topic: 'filtered-events', fromBeginning: false });

  // Windowed aggregation
  const windows = new Map();
  const windowSize = 60000;  // 1 minute

  await consumer.run({
    eachMessage: async ({ message }) => {
      const event = JSON.parse(message.value.toString());
```

```
      const timestamp = new Date(event.timestamp).getTime();
      const windowStart = Math.floor(timestamp / windowSize) * windowSize;
      const key = `${event.pageCategory}:${windowStart}`;

      if (!windows.has(key)) {
        windows.set(key, {
          category: event.pageCategory,
          windowStart: new Date(windowStart).toISOString(),
          viewCount: 0,
          totalDuration: 0
        });
      }

      const window = windows.get(key);
      window.viewCount++;
      window.totalDuration += event.duration;

      // Publish aggregated data
      await producer.send({
        topic: 'aggregated-metrics',
        messages: [{
          key: event.pageCategory,
          value: JSON.stringify(window)
        }]
      });
    }
  });
}

async detectAnomalies() {
  const consumer = this.kafka.consumer({ groupId: 'anomaly-detector' });
  const producer = this.kafka.producer();

  await consumer.connect();
  await producer.connect();
  await consumer.subscribe({ topic: 'aggregated-metrics', fromBeginning: false });

  const baselines = new Map();  // category -> normal view count

  await consumer.run({
    eachMessage: async ({ message }) => {
      const metrics = JSON.parse(message.value.toString());
      const category = metrics.category;

      // Establish baseline
      if (!baselines.has(category)) {
        baselines.set(category, metrics.viewCount);
```

```
        return;
      }

      const baseline = baselines.get(category);

      // Detect spike (3x normal)
      if (metrics.viewCount > baseline * 3) {
        await producer.send({
          topic: 'anomaly-alerts',
          messages: [{
            value: JSON.stringify({
              category,
              expected: baseline,
              actual: metrics.viewCount,
              severity: 'high',
              timestamp: new Date().toISOString()
            })
          }]
        });

        console.log(`🚨 Anomaly detected in ${category}: ${metrics.viewCount} views (expected ~${baseline})`);
      }

      // Update baseline (moving average)
      baselines.set(category, (baseline * 0.9) + (metrics.viewCount * 0.1));
    }
  });
}

async getPageInfo(pageId) {
  // Simulate page metadata lookup
  const categories = ['tech', 'sports', 'news', 'entertainment'];
  return {
    title: `Page ${pageId}`,
    category: categories[Math.floor(Math.random() * categories.length)]
  };
}
}

// Start pipeline
const pipeline = new RealTimeAnalyticsPipeline();
pipeline.start();

// Data flows through pipeline:
// raw-events → filtered-events → aggregated-metrics → anomaly-alerts
```

Lambda Architecture (Batch + Stream)

Concept: Combine batch and real-time processing



Example:

User views page at 10:00:00

- Speed layer: Shows ~100 views (10:00:05)
- Batch layer: Updates to exact 142 views (overnight)
- User sees approximate data immediately, exact data eventually

Kappa Architecture (Stream Only)

Concept: Everything is a stream, no batch layer

Benefits:

- Simpler architecture (one path)
- Can reprocess by replaying stream
- No batch/speed merge complexity

Use when:

- Data fits in stream retention
- Reprocessing acceptable
- Real-time processing sufficient

Real-World Examples

Example 1: E-Commerce Order System

```javascript
```

```
// Event-driven order processing

// Events
const events = {
  ORDER_PLACED: 'ORDER_PLACED',
  PAYMENT_PROCESSED: 'PAYMENT_PROCESSED',
  INVENTORY_RESERVED: 'INVENTORY_RESERVED',
  ORDER_SHIPPED: 'ORDER_SHIPPED',
  ORDER_DELIVERED: 'ORDER_DELIVERED'
};

// Order Service (Publishes events)
class OrderService {
  constructor(eventBus) {
    this.eventBus = eventBus;
  }

  async placeOrder(order) {
    // Store order
    await db.saveOrder(order);

    // Publish event
    await this.eventBus.publish(events.ORDER_PLACED, {
      orderId: order.id,
      userId: order.userId,
      items: order.items,
      total: order.total,
      timestamp: new Date().toISOString()
    });

    return order;
  }
}

// Payment Service (Subscribes to ORDER_PLACED)
class PaymentService {
  constructor(eventBus) {
    this.eventBus = eventBus;

    // Subscribe to events
    this.eventBus.subscribe(events.ORDER_PLACED, (event) => {
      this.processPayment(event);
    });
  }

  async processPayment(orderEvent) {
```

```
    console.log('Processing payment for order:', orderEvent.orderId);

    try {
      // Process payment
      const payment = await this.chargeCustomer(orderEvent.userId, orderEvent.total);

      // Publish success event
      await this.eventBus.publish(events.PAYMENT_PROCESSED, {
        orderId: orderEvent.orderId,
        paymentId: payment.id,
        amount: orderEvent.total,
        timestamp: new Date().toISOString()
      });

    } catch (error) {
      // Publish failure event
      await this.eventBus.publish('PAYMENT_FAILED', {
        orderId: orderEvent.orderId,
        error: error.message,
        timestamp: new Date().toISOString()
      });
    }
  }

  async chargeCustomer(userId, amount) {
    // Simulate payment processing
    await new Promise(resolve => setTimeout(resolve, 1000));
    return { id: 'pay-' + Math.random(), amount };
  }
}

// Inventory Service (Subscribes to PAYMENT_PROCESSED)
class InventoryService {
  constructor(eventBus) {
    this.eventBus = eventBus;

    this.eventBus.subscribe(events.PAYMENT_PROCESSED, (event) => {
      this.reserveInventory(event);
    });
  }

  async reserveInventory(paymentEvent) {
    console.log('Reserving inventory for order:', paymentEvent.orderId);

    // Get order details
    const order = await db.getOrder(paymentEvent.orderId);
```

```
    // Reserve inventory
    for (const item of order.items) {
      await db.updateInventory(item.productId, -item.quantity);
    }

    // Publish event
    await this.eventBus.publish(events.INVENTORY_RESERVED, {
      orderId: paymentEvent.orderId,
      items: order.items,
      timestamp: new Date().toISOString()
    });
  }
}

// Shipping Service (Subscribes to INVENTORY_RESERVED)
class ShippingService {
  constructor(eventBus) {
    this.eventBus = eventBus;

    this.eventBus.subscribe(events.INVENTORY_RESERVED, (event) => {
      this.createShipment(event);
    });
  }

  async createShipment(inventoryEvent) {
    console.log('Creating shipment for order:', inventoryEvent.orderId);

    // Create shipping label
    const shipment = await this.generateLabel(inventoryEvent.orderId);

    // Publish event
    await this.eventBus.publish(events.ORDER_SHIPPED, {
      orderId: inventoryEvent.orderId,
      trackingNumber: shipment.trackingNumber,
      timestamp: new Date().toISOString()
    });
  }

  async generateLabel(orderId) {
    await new Promise(resolve => setTimeout(resolve, 500));
    return { trackingNumber: 'TRACK-' + Math.random() };
  }
}

// Email Service (Subscribes to multiple events)
class EmailService {
  constructor(eventBus) {
```

```
    this.eventBus = eventBus;

    // Subscribe to multiple event types
    this.eventBus.subscribe(events.ORDER_PLACED, this.sendOrderConfirmation.bind(this));
    this.eventBus.subscribe(events.ORDER_SHIPPED, this.sendShippingNotification.bind(this));
    this.eventBus.subscribe('PAYMENT_FAILED', this.sendPaymentFailure.bind(this));
  }

  async sendOrderConfirmation(event) {
    console.log('📧 Sending order confirmation for:', event.orderId);
    // Send email...
  }

  async sendShippingNotification(event) {
    console.log('📧 Sending shipping notification, tracking:', event.trackingNumber);
    // Send email...
  }

  async sendPaymentFailure(event) {
    console.log('📧 Sending payment failure notification for:', event.orderId);
    // Send email...
  }
}

// Event Bus (using Kafka)
class KafkaEventBus {
  constructor() {
    this.kafka = new Kafka({
      clientId: 'event-bus',
      brokers: ['localhost:9092']
    });
    this.producer = this.kafka.producer();
    this.subscribers = new Map();  // topic -> [callbacks]
  }

  async connect() {
    await this.producer.connect();
  }

  async publish(eventType, data) {
    await this.producer.send({
      topic: eventType,
      messages: [{
        value: JSON.stringify(data),
        timestamp: Date.now().toString()
      }]
    });
```

```
    console.log(`Event published: ${eventType}`);
  }

  async subscribe(eventType, callback) {
    if (!this.subscribers.has(eventType)) {
      this.subscribers.set(eventType, []);

      // Create consumer for this event type
      const consumer = this.kafka.consumer({
        groupId: `${eventType}-subscribers`
      });

      await consumer.connect();
      await consumer.subscribe({ topic: eventType, fromBeginning: false });

      await consumer.run({
        eachMessage: async ({ message }) => {
          const data = JSON.parse(message.value.toString());

          // Call all subscribers
          const callbacks = this.subscribers.get(eventType);
          for (const cb of callbacks) {
            try {
              await cb(data);
            } catch (error) {
              console.error(`Error in subscriber for ${eventType}:`, error);
            }
          }
        }
      });
    }

    this.subscribers.get(eventType).push(callback);
  }
}

// Wire everything together
async function main() {
  const eventBus = new KafkaEventBus();
  await eventBus.connect();

  // Initialize services
  const orderService = new OrderService(eventBus);
  const paymentService = new PaymentService(eventBus);
  const inventoryService = new InventoryService(eventBus);
  const shippingService = new ShippingService(eventBus);
```

```
  const emailService = new EmailService(eventBus);

  // Place order
  await orderService.placeOrder({
    id: 'order-123',
    userId: 'user-456',
    items: [
      { productId: 'prod-1', quantity: 2, price: 50 }
    ],
    total: 100
  });

  // Output (events flow through system):
  // Event published: ORDER_PLACED
  // Processing payment for order: order-123
  // 📧 Sending order confirmation for: order-123
  // Event published: PAYMENT_PROCESSED
  // Reserving inventory for order: order-123
  // Event published: INVENTORY_RESERVED
  // Creating shipment for order: order-123
  // Event published: ORDER_SHIPPED
  // 📧 Sending shipping notification, tracking: TRACK-...
}

main();
```

Example 2: Real-Time User Activity Tracking

javascript

```
// Track and analyze user behavior in real-time

class UserActivityTracker {
  constructor() {
    this.kafka = new Kafka({
      clientId: 'activity-tracker',
      brokers: ['localhost:9092']
    });
  }

  // Collect activity events
  async trackActivity(activity) {
    const producer = this.kafka.producer();
    await producer.connect();

    await producer.send({
      topic: 'user-activity',
      messages: [{
        key: activity.userId,  // Partition by user
        value: JSON.stringify({
          userId: activity.userId,
          action: activity.action,
          page: activity.page,
          timestamp: new Date().toISOString(),
          metadata: activity.metadata
        })
      }]
    });

    await producer.disconnect();
  }

  // Real-time session analysis
  async analyzeUserSessions() {
    const consumer = this.kafka.consumer({ groupId: 'session-analyzer' });
    const producer = this.kafka.producer();

    await consumer.connect();
    await producer.connect();
    await consumer.subscribe({ topic: 'user-activity', fromBeginning: false });

    const sessions = new Map();  // userId -> current session
    const SESSION_TIMEOUT = 30 * 60 * 1000;  // 30 minutes

    await consumer.run({
      eachMessage: async ({ message }) => {
```

```
const activity = JSON.parse(message.value.toString());
const userId = activity.userId;
const now = new Date(activity.timestamp).getTime();

// Get or create session
if (!sessions.has(userId)) {
  sessions.set(userId, {
    userId,
    sessionId: `session-${userId}-${now}`,
    startTime: activity.timestamp,
    lastActivity: activity.timestamp,
    actions: [],
    pages: new Set()
  });
}

const session = sessions.get(userId);
const lastActivity = new Date(session.lastActivity).getTime();

// Check if session timeout
if (now - lastActivity > SESSION_TIMEOUT) {
  // End current session
  await this.endSession(producer, session);

  // Start new session
  sessions.set(userId, {
    userId,
    sessionId: `session-${userId}-${now}`,
    startTime: activity.timestamp,
    lastActivity: activity.timestamp,
    actions: [],
    pages: new Set()
  });
}

// Update session
session.actions.push(activity.action);
session.pages.add(activity.page);
session.lastActivity = activity.timestamp;

// Detect interesting patterns
if (session.actions.length === 10) {
  console.log(`User ${userId} has 10 actions in current session`);
}

if (session.pages.size >= 5) {
  console.log(`User ${userId} viewed 5+ different pages`);
```

```
        // Engaged user! Could trigger recommendation
      }
    }
  });

  // Periodic session cleanup
  setInterval(async () => {
    const now = Date.now();
    for (const [userId, session] of sessions) {
      const lastActivity = new Date(session.lastActivity).getTime();
      if (now - lastActivity > SESSION_TIMEOUT) {
        await this.endSession(producer, session);
        sessions.delete(userId);
      }
    }
  }, 60000);  // Check every minute
}

async endSession(producer, session) {
  const duration = new Date(session.lastActivity) - new Date(session.startTime);

  // Publish session summary
  await producer.send({
    topic: 'user-sessions',
    messages: [{
      key: session.userId,
      value: JSON.stringify({
        sessionId: session.sessionId,
        userId: session.userId,
        startTime: session.startTime,
        endTime: session.lastActivity,
        duration: duration,
        actionCount: session.actions.length,
        pageCount: session.pages.size,
        actions: session.actions
      })
    }]
  });

  console.log(`Session ended for user ${session.userId}: ${session.actions.length} actions, ${duration}ms`);
}
}

// Real-time personalization
class PersonalizationEngine {
  constructor() {
    this.userPreferences = new Map();
```

```
}

async updatePreferences() {
  const consumer = this.kafka.consumer({ groupId: 'personalization' });
  await consumer.connect();
  await consumer.subscribe({ topic: 'user-activity', fromBeginning: false });

  await consumer.run({
    eachMessage: async ({ message }) => {
      const activity = JSON.parse(message.value.toString());
      const userId = activity.userId;

      // Track user interests
      if (!this.userPreferences.has(userId)) {
        this.userPreferences.set(userId, {
          categories: new Map(),
          recentPages: []
        });
      }

      const prefs = this.userPreferences.get(userId);

      // Extract category from page
      const category = this.extractCategory(activity.page);
      if (category) {
        prefs.categories.set(
          category,
          (prefs.categories.get(category) || 0) + 1
        );
      }

      // Track recent pages
      prefs.recentPages.push(activity.page);
      if (prefs.recentPages.length > 20) {
        prefs.recentPages.shift();
      }

      // Update recommendation model in real-time
      if (prefs.categories.size >= 3) {
        const topCategory = [...prefs.categories.entries()]
          .sort((a, b) => b[1] - a[1])[0][0];

        console.log(`User ${userId} top interest: ${topCategory}`);
        // Could trigger product recommendations
      }
    }
  });
```

```
  }

  extractCategory(page) {
    // Extract category from URL
    const match = page.match(/\/category\/([^\/]+)/);
    return match ? match[1] : null;
  }

  getRecommendations(userId) {
    const prefs = this.userPreferences.get(userId);
    if (!prefs) return [];

    // Get top categories
    const topCategories = [...prefs.categories.entries()]
      .sort((a, b) => b[1] - a[1])
      .slice(0, 3)
      .map(([cat, _]) => cat);

    return topCategories;
  }
}
```

Example 3: Real-Time Monitoring Dashboard

javascript

```
// Real-time metrics aggregation

class MetricsAggregator {
  constructor() {
    this.kafka = new Kafka({
      clientId: 'metrics-aggregator',
      brokers: ['localhost:9092']
    });

    this.metrics = {
      requestsPerSecond: 0,
      avgResponseTime: 0,
      errorRate: 0,
      activeUsers: new Set()
    };
  }

  async aggregateMetrics() {
    const consumer = this.kafka.consumer({ groupId: 'metrics-aggregator' });
    const producer = this.kafka.producer();

    await consumer.connect();
    await producer.connect();
    await consumer.subscribe({ topic: 'api-logs', fromBeginning: false });

    // Tumbling window: 1 second
    let windowStart = Date.now();
    let windowEvents = [];

    await consumer.run({
      eachMessage: async ({ message }) => {
        const log = JSON.parse(message.value.toString());
        const now = Date.now();

        // Check if window expired
        if (now - windowStart >= 1000) {
          // Process completed window
          await this.processWindow(producer, windowEvents);

          // Start new window
          windowStart = now;
          windowEvents = [];
        }

        // Add to current window
        windowEvents.push(log);
```

```
    }
  });
}

async processWindow(producer, events) {
  if (events.length === 0) return;

  // Calculate metrics for this window
  const metrics = {
    timestamp: new Date().toISOString(),
    requestCount: events.length,
    avgResponseTime: events.reduce((sum, e) => sum + e.responseTime, 0) / events.length,
    errorCount: events.filter(e => e.statusCode >= 400).length,
    uniqueUsers: new Set(events.map(e => e.userId)).size,

    // Breakdown by endpoint
    endpoints: {}
  };

  // Group by endpoint
  events.forEach(event => {
    if (!metrics.endpoints[event.endpoint]) {
      metrics.endpoints[event.endpoint] = {
        count: 0,
        avgResponseTime: 0,
        errors: 0
      };
    }

    const ep = metrics.endpoints[event.endpoint];
    ep.count++;
    ep.avgResponseTime += event.responseTime;
    if (event.statusCode >= 400) ep.errors++;
  });

  // Calculate averages
  for (const endpoint in metrics.endpoints) {
    metrics.endpoints[endpoint].avgResponseTime /= metrics.endpoints[endpoint].count;
  }

  // Publish aggregated metrics
  await producer.send({
    topic: 'metrics-aggregated',
    messages: [{
      value: JSON.stringify(metrics)
    }]
  });
```

```
    console.log(`Window metrics: ${metrics.requestCount} req/s, ${metrics.avgResponseTime.toFixed(2)}ms avg`);

    // Check for anomalies
    if (metrics.errorCount > metrics.requestCount * 0.05) {
      console.log(`🚨 High error rate: ${(metrics.errorCount / metrics.requestCount * 100).toFixed(2)}%`);
    }
  }
}

// Dashboard updater (WebSocket to frontend)
class DashboardUpdater {
  constructor(websocketServer) {
    this.wss = websocketServer;
    this.kafka = new Kafka({
      clientId: 'dashboard-updater',
      brokers: ['localhost:9092']
    });
  }

  async streamToClients() {
    const consumer = this.kafka.consumer({ groupId: 'dashboard' });
    await consumer.connect();
    await consumer.subscribe({ topic: 'metrics-aggregated', fromBeginning: false });

    await consumer.run({
      eachMessage: async ({ message }) => {
        const metrics = JSON.parse(message.value.toString());

        // Broadcast to all connected dashboard clients
        this.wss.clients.forEach(client => {
          if (client.readyState === WebSocket.OPEN) {
            client.send(JSON.stringify({
              type: 'metrics_update',
              data: metrics
            }));
          }
        });

        console.log('Metrics pushed to dashboard clients');
      }
    });
  }
}
```

Key Takeaways

1. **Event Sourcing:**
   - Store all changes as events (immutable log)
   - Rebuild state by replaying events
   - Complete audit trail
   - Time travel capability
   - Use snapshots for performance
2. **CQRS:**
   - Separate read and write models
   - Optimize each for its purpose
   - Write: Normalized, ACID
   - Read: Denormalized, fast queries
   - Eventual consistency acceptable
3. **Event Streaming:**
   - Kafka as event log
   - Multiple consumers at different offsets
   - Can replay events
   - Real-time processing
   - Lambda/Kappa architectures
4. **Real-Time Processing:**
   - Stateless transformations (filter, map)
   - Stateful aggregations (count, sum)
   - Windowed operations (time-based)
   - Pattern detection (fraud, anomalies)
   - Stream joins

Practice Problems

1. Design an event-sourced banking system. What events would you store? How would you handle snapshots?
2. Design a real-time analytics dashboard for a website. What architecture would you use?
3. Compare event sourcing + CQRS vs traditional CRUD for an e-commerce platform. Which would you choose and why?

4. Design a fraud detection system that analyzes transaction events in real-time.

Next Chapter Preview

In Chapter 12, we'll explore **High Availability**:

- Achieving the "nines" (99.9%, 99.99%, 99.999%)
- Redundancy and replication
- Failover mechanisms
- Active-active vs active-passive
- Disaster recovery

Ready to continue?