

# Happen

# Welcome

Welcome to Happen, where we believe in the power of simplicity!

We're excited you're here to explore a framework that turns conventional wisdom on its head. Instead of drowning you in abstractions and boilerplate, Happen gives you just two fundamental building blocks—Nodes and Events—that combine to create systems of surprising power and flexibility. Whether you're building a straightforward automation pipeline or an intricate multi-agent ecosystem, you'll find that Happen's minimalist approach makes complex problems suddenly manageable.

Our philosophy is simple: the most elegant solutions emerge not from adding complexity, but from discovering the right minimal abstractions that let the magic happen naturally. As you dive into these docs, you'll see how a handful of simple concepts can unlock an entire universe of possibilities. So take a deep breath, let go of framework complexity, and welcome to a world where less truly is more.

## Quick Links



[Basics](#)

Learn



[Getting Started](#)

Dive right in

# Getting Started

# What is Happen?

Happen is a framework for building agent-based systems founded on a philosophy of simplicity. Unlike conventional frameworks that burden developers with complex abstractions, Happen provides just two fundamental building blocks that can be composed to create systems ranging from simple pipelines to complex, adaptive multi-agent ecosystems.

## Core Philosophy

At its core, Happen distills agent-based systems down to their essence, recognizing that true power emerges from simplicity rather than complexity. The framework's philosophy embraces several key principles:

- **Radical Simplicity:** Only include what's absolutely essential; solve specific needs with specialized nodes rather than general framework features
- **Pure Causality:** Everything in a Happen system happens because of something else, creating natural causal chains
- **Decentralized Intelligence:** Smart systems emerge from simple nodes making local decisions
- **Composable Simplicity:** Complex behavior emerges from composing simple, understandable parts
- **Emergence Over Prescription:** Let patterns emerge naturally rather than forcing specific structural approaches

This minimalist approach creates a framework that is both accessible to newcomers and powerful enough for experts, allowing developers to focus on solving domain problems rather than battling framework complexities.

## The Foundation: Nodes and Events

At its foundation lie just two primitives:

1. **Nodes** - Independent, autonomous components that process and respond to information
2. **Events** - Structured messages that transport data and intentions between Nodes

These two simple concepts combine to create systems of surprising power and flexibility. Nodes act as the primary actors in a Happen system - independent, autonomous components that can receive and process events, maintain internal state, and emit new events. Events are structured messages that flow between nodes, representing meaningful occurrences in the domain rather than simple function calls.

## Pure Causality: The Secret to Happen's Power

The most distinctive aspect of Happen is its embrace of pure causality as its organizing principle. This approach eliminates entire categories of complexity found in other systems.

In Happen:

- Events can only be created when their prerequisites exist
- Every event contains references to its causal predecessors
- Each event has a unique identifier that gets referenced by its dependents
- These references form a complete causal web that defines system behavior

This means that knowledge about the system is encoded directly in event flow patterns. Nodes don't need schema definitions or formal contracts to understand how to interact - they simply need to observe the event flow.

When a node receives an event, it automatically knows:

- What event caused this one

- What transaction or process it's part of
- Which node originated the event

The entire history and context are embedded in the event itself.

## Runtime Transparency: Direct Access to Your Environment

Happen embodies simplicity not only in its design but in its relationship with the underlying runtime environment. Unlike conventional frameworks that build abstraction layers over runtime capabilities, Happen takes a fundamentally different approach:

Happen is simply a coordination layer for events. The runtime belongs to your code.

This means:

- No framework-specific abstractions between your code and the runtime
- Direct use of runtime capabilities without performance penalties
- Immediate access to new runtime features without waiting for framework updates
- Seamless integration with the broader ecosystem for your runtime

## Why Happen? The Power of Less

Happen's minimalist approach offers profound advantages:

1. **Cognitive Accessibility:** The entire framework can be understood in minutes, eliminating the steep learning curves associated with complex agent frameworks

2. **Unbounded Flexibility:** The simple primitives can be composed in countless ways, enabling virtually any interaction pattern
3. **Natural Scalability:** Systems grow organically without architectural reimagining, scaling from tiny applications to vast distributed networks
4. **Inherent Resilience:** The decentralized, event-driven nature creates natural recovery patterns and adaptation to changing conditions
5. **Self-Documentation:** The causal web of events documents system behavior automatically

Happen demonstrates that remarkable power can emerge from intentional simplicity. By providing just enough structure through two core concepts, it enables developers to create systems that are both powerful and comprehensible.

As you begin building with Happen, remember that the framework's power comes not from complex features or abstractions, but from how its minimal primitives combine to create emergent behaviors. Start simple, observe how events flow through your system, and let the natural patterns guide your development.

# Runtime Philosophy

Happen embodies simplicity not only in its design but in its relationship with the underlying runtime environment. Unlike conventional frameworks that build abstraction layers over runtime capabilities, Happen takes a fundamentally different approach:

Happen is simply a coordination layer for events. The runtime belongs to your code.

This approach has profound implications for how you build systems with Happen.

## Direct Runtime Access

With Happen, there are no framework-specific abstractions standing between your code and the runtime:

```
// A Happen node handling file operations
fileNode.on('save-document', async event => {
  const { path, content } = event.payload;
  try {
    // Direct use of runtime capabilities
    // No "Happen way" to do this - just use the runtime
    await Bun.write(path, content);

    // Return success result
    return {
      status: 'success',
      path
    };
  } catch (error) {
    // Return error result
    return {
      status: 'error',
      reason: error.message
    };
  }
})
```



```
    5  
  });
```

This example demonstrates a crucial aspect of Happen's philosophy: the framework doesn't try to abstract, wrap, or "improve" the runtime's capabilities. Your event handlers are simply JavaScript functions with full, direct access to whatever runtime they're executing in.

## The Invisible Framework

Happen is designed to be virtually invisible from a runtime perspective:

```
// Creating a web server node  
httpNode.on('start-server', event => {  
  const { port } = event.payload;  
  
  // Direct use of Bun's HTTP server capabilities  
  const server = Bun.serve({  
    port,  
    fetch(request) {  
      // Convert HTTP requests to events  
      const requestEvent = {  
        type: 'http-request',  
        payload: {  
          url: request.url,  
          method: request.method,  
          headers: Object.fromEntries(request.headers.entries()),  
          body: request.body  
        }  
      };  
  
      // Process through Happen's event system  
      const responseEvent = httpNode.processSync(requestEvent);  
  
      // Convert response event back to HTTP  
      return new Response(  
        responseEvent.payload.body,  
        {  
          status: responseEvent.payload.status,  
          headers: responseEvent.payload.headers  
        }  
      );  
    }  
  });
```

```
    );  
  }  
});  
  
// Store server instance for later management  
httpNode.server = server;  
  
return { status: 'started', port };  
});
```

In this example, the node creates and manages a web server using the runtime's native capabilities. Happen simply provides the event-based coordination layer, not the HTTP functionality itself.

## "Glue with Superpowers"

This runtime philosophy exemplifies Happen's role as "glue with superpowers":

1. **It's Just Glue:** Happen connects different pieces of functionality without dictating how that functionality is implemented
2. **Your Code, Your Runtime:** Your event handlers have complete freedom to use the runtime however they need
3. **No Hidden Magic:** There's no framework-specific way to access runtime features - you use them directly
4. **No Learning Curve:** If you know how to use the runtime, you already know how to use it with Happen

## The Benefits of Runtime Transparency

This direct relationship between your code and the runtime offers several key advantages:

### 1. Zero Overhead

Since Happen doesn't abstract the runtime, there's no performance penalty for accessing runtime capabilities:

```
// No performance penalty for runtime access
databaseNode.on('query-users', async event => {
  const { filter } = event.payload;

  // Direct database query using runtime capabilities
  // No layers of abstraction adding overhead
  const statement = db.prepare(`
    SELECT * FROM users
    WHERE name LIKE ?
  `);
  const users = statement.all(`%${filter}%`);

  return { users };
});
```

## 2. Full Runtime Capabilities

You have access to everything the runtime offers, not just what the framework developers thought to expose:

```
// Using advanced runtime features
processingNode.on('process-image', async event => {
  const { imagePath, options } = event.payload;

  // Access to cutting-edge runtime features
  // as soon as they're available in the runtime
  const imageData = await Bun.file(imagePath).arrayBuffer();
  const processed = await ImageProcessor.process(imageData,
options);
  await Bun.write(`${imagePath}.processed`, processed);

  return { status: 'completed', outputPath:
`${imagePath}.processed` };
});
```

### 3. Runtime Evolution

As the runtime evolves with new capabilities, they're immediately available without waiting for framework updates:

```
// Immediate access to new runtime features
node.on('use-new-feature', event => {
  // When the runtime adds new capabilities,
  // you can use them immediately without framework updates
  const result = Bun.newCapability(event.payload);

  return { result };
});
```

### 4. Ecosystem Compatibility

Your Happen code works seamlessly with the broader ecosystem for your runtime:

```
// Seamless integration with ecosystem libraries
analyticsNode.on('analyze-data', async event => {
  const { dataset } = event.payload;

  // Use any ecosystem library directly
  // No need for framework-specific adapters or wrappers
  const analysis = await performAnalysis(dataset);

  return { results: analysis };
});

// Helper function using ecosystem libraries
async function performAnalysis(dataset) {
  // Direct use of ecosystem libraries
  // No "Happen way" to integrate libraries
  const results = await someAnalyticsLibrary.process(dataset);
  return results;
}
```

# Consistent Philosophy Across Environments

This runtime philosophy remains consistent whether you're using Bun, Node.js, Deno, or browsers:

```
// The same approach works across different runtimes
function createFileNode(id) {
  const node = createNode(id);

  // Handler implementation uses runtime-appropriate APIs
  // but the Happen code remains the same
  node.on('read-file', async event => {
    const { path } = event.payload;

    try {
      // In Bun
      const content = await Bun.file(path).text();

      // In Node.js
      // const content = await fs.promises.readFile(path, 'utf8');

      // In Deno
      // const content = await Deno.readTextFile(path);

      // In browser
      // const response = await fetch(path);
      // const content = await response.text();

      return { content };
    } catch (error) {
      return { error: error.message };
    }
  });

  return node;
}
```

While the specific API calls differ between environments, Happen's relationship with the runtime remains the same: it simply provides event coordination while giving your code direct access to the runtime.

# The Power of Stepping Back

By stepping back and focusing solely on event coordination, Happen achieves something rare in frameworks: it gets out of your way. The framework doesn't try to be the center of your application or dictate how you should use the runtime. It simply provides the minimal structure needed for nodes to communicate through events, then lets your code take center stage.

This approach creates a unique developer experience:

1. **Low Cognitive Load:** There's no need to learn framework-specific ways to access runtime features
2. **Maximum Flexibility:** Your code can use the runtime however it needs without framework constraints
3. **Future-Proof:** As runtimes evolve, your code can immediately leverage new capabilities
4. **True Simplicity:** The framework does one thing well (event coordination) and leaves the rest to your code

Happen's runtime philosophy embodies its commitment to simplicity. By focusing solely on event coordination and giving your code direct access to the runtime, Happen creates a framework that is both minimal and powerful.

This approach recognizes that the most powerful frameworks are often those that do less, not more. By stepping back and letting your code work directly with the runtime, Happen achieves a rare balance: it provides enough structure to create sophisticated event-driven systems while imposing minimal constraints on how those systems are implemented.

The result is a framework that truly embodies its name: it helps things happen without dictating how they happen.

# Quick Start

Welcome to Happen, a framework for building agentic systems founded on a philosophy of simplicity. Unlike conventional frameworks that burden developers with complex abstractions, Happen provides just two fundamental building blocks that can be composed to create systems ranging from simple pipelines to complex, adaptive multi-agent ecosystems.

Happen distills agent-based systems down to their essence, recognizing that true power emerges from simplicity rather than complexity. At its foundation lie just two primitives:

1. **Nodes** - Independent, autonomous components that process and respond to information
2. **Events** - Structured messages that transport data and intentions between Nodes

This deliberate minimalism creates a framework that is both accessible to newcomers and powerful enough for experts, allowing developers to focus on solving domain problems rather than battling framework complexities.

## Setup and Initialization

To get started with Happen, first initialize the framework with your desired configuration:

```
// Initialize Happen with NATS as the communication fabric
const happen = initializeHappen({
  // NATS configuration
  nats: {
    // Connection configuration
    connection: {
      // Server environment (direct NATS)
      server: {
        servers: ['nats://localhost:4222'],
```

```
        jetstream: true
      },
      // Browser environment (WebSocket)
      browser: {
        servers: ['wss://localhost:8443'],
        jetstream: true
      }
    }
  }
});

// Extract the createNode function
const { createNode } = happen;
```

For simpler use cases, you can use the default configuration:

```
// Use default configuration with local NATS server
const { createNode } = initializeHappen();
```

## Pure Causality: The Secret to Happen's Power

The most distinctive aspect of Happen is its embrace of pure causality as its organizing principle. This approach eliminates entire categories of complexity found in other systems.

In Happen:

- Events can only be created when their prerequisites exist
- Every event contains references to its causal predecessors
- Each event has a unique identifier that gets referenced by its dependents
- These references form a complete causal web that defines system behavior

This means that knowledge about the system is encoded directly in event flow patterns. Nodes don't need schema definitions or formal contracts to understand how to interact - they simply need to observe the event flow.



# The Event Continuum: Programmable Flow Control

One of Happen's most powerful features is the "Event Continuum" model, which gives you complete control over event handling and flow transitions. Unlike typical event handlers that simply execute and complete, Happen handlers can control what happens next:

```
// Register an event handler - the first step in processing
orderNode.on("process-order", function validateOrder(event,
context) {
  // Validate the order
  const validation = validateOrderData(event.payload);

  if (!validation.valid) {
    // Return a value to complete the flow with an error
    return {
      success: false,
      reason: "validation-failed",
      errors: validation.errors
    };
  }

  // Store validation result in context for later steps
  context.validation = validation;

  // Return the next function to execute - controlling the
  transition
  return processPayment;
});

// The next function in the flow
function processPayment(event, context) {
  // Access data from previous step through context
  const validatedOrder = context.validation;

  // Process payment
  const paymentResult = processTransaction(validatedOrder);

  if (!paymentResult.success) {
    // Return a value to complete with failure
    return {
```

```
        success: false,
        reason: "payment-failed"
    };
}

// Store payment result in context
context.payment = paymentResult;

// Return the next function to transition to
return createShipment;
}

// Final step in the flow
function createShipment(event, context) {
    // Access data from previous steps
    const { payment, validation } = context;

    // Create shipment
    const shipment = generateShipment(validation.address);

    // Return final result to complete the flow
    return {
        success: true,
        orderId: validation.orderId,
        paymentId: payment.transactionId,
        trackingNumber: shipment.trackingNumber
    };
}
```

This functional flow approach means:

1. **Explicit Transitions:** You control what happens next by returning either a value (to complete) or a function (to continue)
2. **Shared Context:** Information flows between steps through the context object
3. **Natural Branching:** Flow can branch based on conditions without complex routing rules
4. **Composable Workflows:** Complex flows emerge from simple, focused functions

The Event Continuum model enables remarkably clean and readable code for complex workflows while maintaining full control over transitions.

## The Causal Event Web

When a node receives this event, it automatically knows:

```
// An event naturally references its causal predecessors
{
  type: 'order-shipped',
  payload: {
    orderId: 'ORD-123',
    trackingNumber: 'TRK-456'
  },
  context: {
    // ...
    causal: {
      id: 'evt-789', // Unique identifier
      sender: 'shipping-node', // Origin node
      causationId: 'evt-456', // Direct cause (payment
confirmation)
      correlationId: 'order-123' // Overall transaction
    }
  }
}
```

- This event was caused by event 'evt-456'
- It's part of the transaction 'order-123'
- It originated from 'shipping-node'

The entire history and context are embedded in the event itself.

This pure causality model means:

- No need for schema registries
- No explicit contract definitions

- No capability discovery protocols
- No separate identity systems

The system self-describes through its natural operation, with each event reinforcing the web of causality that defines how components interact.

## Building Your First Happen System

Let's create a simple order processing system to demonstrate Happen's principles in action:

First, we'll create autonomous nodes to handle different aspects of our order process:

```
// Create independent nodes
const orderNode = createNode('order-service');
const inventoryNode = createNode('inventory-service');
const paymentNode = createNode('payment-service');
const shippingNode = createNode('shipping-service');
```

Next, we'll define how each node responds to events using the Event Continuum:

```
// Order node handles order creation with flow control
orderNode.on('create-order', function validateOrder(event, context)
{
  // Validate order data
  const validation = validateOrderData(event.payload);

  if (!validation.valid) {
    return {
      success: false,
      errors: validation.errors
    };
  }

  // Generate order ID
```

```

const orderId = generateOrderId();

// Transition to inventory check, passing data through context
context.orderData = {
  id: orderId,
  items: event.payload.items,
  customer: event.payload.customer
};

return checkInventory;
});

// Additional flow functions defined here...

```

Finally, we'll initialize the system and process an example order:

```

function startOrderSystem() {
  console.log('Order processing system initialized and ready');

  // Broadcast an order creation event to start the process
  orderNode.broadcast({
    type: 'create-order',
    payload: {
      customer: {
        id: 'cust-123',
        name: 'Jane Doe',
        email: 'jane@example.com',
        address: {
          street: '123 Main St',
          city: 'Anytown',
          zip: '12345'
        }
      },
      items: [
        { id: 'prod-a1', quantity: 2 },
        { id: 'prod-b7', quantity: 1 }
      ]
    }
  });

  // The system now autonomously processes this event through the
  // flow:
  // 1. validateOrder function runs and transitions to
  // 2. checkInventory function runs and transitions to
  // 3. createOrder function runs and transitions to
  // 4. ...

```

```
checkInventory
  // 2. checkInventory function runs and transitions to
  processPayment
  // 3. processPayment function runs and transitions to
  createShipment
  // 4. createShipment function completes the flow and returns a
  result
  // All steps share context and control their own transitions
}

// Start the system
startOrderSystem();
```

## Communication Patterns

Happen provides two communication patterns that can be composed to create any interaction model:

### 1. System-wide Broadcasting

Events are sent to all nodes in the system:

```
// Weather service broadcasts a system-wide alert
weatherNode.broadcast({
  type: 'severe-weather',
  payload: {
    condition: 'hurricane',
    region: 'Gulf Coast'
  }
});

// Any node can respond
emergencyNode.on('severe-weather', event => {
  deployEmergencyTeams(event.payload.region);
});
```

### 2. Direct Communication

## Point-to-point request-response interactions:

```
// Direct request with awaited response
const response = await orderNode.send(paymentNode, {
  type: 'process-payment',
  payload: {
    orderId: 'ORD-28371',
    amount: 129.99
  }
});

// Handle the response
if (response.success) {
  completeOrder(response.orderId);
}
```

The receiving node knows to return a value because the event context automatically includes information about the sender and whether a response is expected:

```
// Payment node handling direct communication
paymentNode.on('process-payment', (event, context) => {
  // The context contains information about the sender and request
  console.log(context.sender); // 'order-service'
  console.log(context.expectsResponse); // true

  // Process the payment
  const result = processPayment(event.payload.orderId,
    event.payload.amount);

  // Simply returning a value sends it back to the requester
  return {
    success: result.success,
    transactionId: result.success ? result.transactionId : null,
    status: result.success ? 'approved' : 'declined',
    message: result.message
  };
});
```

This simple mechanism means that nodes don't need to know anything

special about handling direct communication - they just return values from their handlers as usual. The Happen framework takes care of routing the response back to the sender when a response is expected.

## Distributed Operation

With NATS as the underlying messaging fabric, your Happen system automatically works across different processes, machines, and environments. The same code works whether nodes are:

- In the same process
- In different processes on the same machine
- On different machines in the same network
- In different environments (server, browser, edge)

No special configuration is needed - nodes communicate transparently across boundaries.

## Next Steps

As you build with Happen, remember these key principles:

1. **Start Simple:** Begin with a few nodes that have clear responsibilities
2. **Focus on Events:** Design meaningful events that capture domain concepts
3. **Control Transitions:** Use function returns to control your flow
4. **Observe Patterns:** Let natural workflows emerge from event interactions
5. **Compose, Don't Complicate:** Combine the basic primitives instead of adding complexity
6. **Embrace Causality:** Use the causal event web to reason about your system



The power of Happen comes not from complex features or abstractions, but from how its minimal primitives combine to create emergent behaviors.

Ready to explore more? Continue to the Core Concepts section to deepen your understanding of nodes, events, and the Happen framework.

# Basics

# Core Concepts

Events all the way down.

Happen is built on a minimalist foundation of just two fundamental primitives: Nodes and Events. These primitives combine to create all the power of the framework without unnecessary complexity.

At the infrastructure level, these primitives are powered by NATS, a high-performance messaging system that provides the backbone for Happen's distributed capabilities.

## Nodes: The Foundation of Happen Systems

Nodes are the primary actors in a Happen system. They are independent, autonomous components that can:

- Receive and process events
- Maintain internal state
- Transform state
- Emit new events

Conceptually, nodes can represent anything: services, entities, agents, processes, or even abstract concepts. This flexibility allows Happen systems to model complex domains with remarkable fidelity.

Unlike traditional components that often require elaborate setup and configuration, Happen nodes are intentionally lightweight. They embody a biological inspiration—like cells in an organism or neurons in a brain—simple individual units that collectively create sophisticated behaviors through their interactions.

## Creating a Node

Creating a node in Happen is achieved by calling the `createNode()` function with an identifier and optional config object:

```
const orderProcessor = createNode("order-processor", {});
```

Every node has a unique identity that distinguishes it within the system:

- Identities are cryptographically verifiable
- Nodes act autonomously based on their internal logic
- A node's behavior is determined by how it processes events and transforms state
- Nodes don't need to know about the overall system structure to function

This autonomy means nodes can be developed, tested, and reasoned about independently, dramatically simplifying system development and maintenance.

## Events: The Lifeblood of the System

Events are structured messages that flow between nodes, forming the lifeblood of any Happen system. They embody the principle that communication should be explicit, intentional, and meaningful.

Unlike traditional method calls or function invocations, events in Happen represent meaningful occurrences in the domain—things that have happened that might be of interest to other parts of the system. This perspective shift encourages developers to model their systems in terms of meaningful domain events rather than technical procedures.

Under the hood, Happen uses NATS to transport these events between nodes, benefiting from its high-performance, reliable messaging capabilities.

## Event Anatomy

Events consist of three fundamental parts:

1. **Type** - A string identifier indicating the event's purpose
2. **Payload** - Domain-specific data relevant to the event
3. **Context** - Causal and system information about the event

```
{
  type: "order-created",
  payload: {
    orderId: "ORD-123",
    items: [
      {
        productId: "PROD-456",
        quantity: 2,
        price: 29.99
      }
    ],
    total: 59.98
  },
  context: {
    causal: {
      id: "evt-789", // Unique event identifier
      sender: "checkout-node", // Node that created the event
      causationId: "evt-456", // Event that caused this one
      correlationId: "txn-123", // Transaction/process identifier
      path: ["user-node", "checkout-node"] // Event path
    },
    // Additional context...
  }
}
```

## Event Processing: The Event Continuum

Nodes process events through a pure functional flow model we call the "Event Continuum." This approach treats event handling as a chain of functions where each function determines what happens next through its return value:

```
// Register an event handler - the entry point to the flow
orderNode.on("process-order", validateOrder);

// First function in the flow
function validateOrder(event, context) {
  // Validate order
  const validation = validateOrderData(event.payload);
  if (!validation.valid) {
    return { success: false, reason: "Invalid order" };
  }

  // Store validation result
  context.validatedOrder = {
    ...event.payload,
    validated: true,
    validatedAt: Date.now()
  };

  // Return the next function to execute
  return processPayment;
}

// Next function in the flow
function processPayment(event, context) {
  // Process payment using data from context
  const paymentResult = processTransaction(event.payload.payment);

  // Store in context
  context.payment = paymentResult;

  if (!paymentResult.success) {
    // Return a value to complete with failure
    return { success: false, reason: "Payment failed" };
  }

  // Return next function
  return createShipment;
}

// Final function returning a result value
function createShipment(event, context) {
  // Create shipment
  const shipment = generateShipment(event.payload.address);

  // Return final result (not a function)
```

```
// Return final result (not a function)
return {
  success: true,
  orderId: event.payload.orderId,
  trackingNumber: shipment.trackingNumber
};
}
```

This pure functional approach offers remarkable power and flexibility:

- Any function can return another function to redirect the flow
- Any function can return a non-function value to complete the flow
- A shared context object allows communication between functions
- Complex workflows emerge naturally from function composition

## State and Persistence

Nodes can maintain state that persists between events. Happen provides a clean, functional approach to accessing and updating state through pattern matching and transformations.

Under the hood, this state is persisted in NATS JetStream's Key-Value store, providing durable, distributed state management without requiring custom persistence implementations.

## Accessing State

```
// Access the entire node state
const orderState = orderNode.state.get();

// Transform state before returning
const orders = orderNode.state.get(state => state.backlog);

// Access and transform state
const correctedBatch = orderNode.state.get(state => {
  // Transform the state
  return state.orders.map(order => ({
```

```

    ...state.orders.map((order) => {
      ...order,
      total: recalculateTotal(order)
    }));
  });
};

```

## Transforming State

You can transform state using a single function approach:

```

// Transform state without pattern matching
orderNode.state.set((state) => {
  // Update transformed state
  return {
    ...state,
    orders: {
      ...state.orders,
      "order-123": {
        ...state.orders["order-123"],
        status: "shipped",
        shippedAt: Date.now()
      }
    }
  };
});

```

For more focused updates, you can create specific transformers:

```

// Define transformer for shipping an order
const shipOrder = (orderId) => (state) => ({
  ...state,
  orders: {
    ...state.orders,
    [orderId]: {
      ...state.orders[orderId],
      status: "shipped",
      shippedAt: Date.now()
    }
  }
});

```



```
// Apply transformer
orderNode.state.set(shipOrder("order-123"));
```

## The Causality Web

Underlying Happen's approach is the concept of pure causality, where every event is part of a causal chain. This is implemented through the metadata in each event:

- `causationId`: References the event that directly caused this one
- `correlationId`: Groups events that are part of the same transaction or process
- `path`: Shows the journey the event has taken through the system

This causal relationship creates a complete web of events that defines the system's behavior without requiring explicit schema definitions, contracts, or capability discovery protocols.

## Uniting Event Processing and State Transformation

What makes Happen truly powerful is how the Event Continuum and state transformations work together:

```
// Process event with the Event Continuum
orderNode.on("update-order-status", (event, context) => {
  // Validate status change
  const validationResult = validateStatusChange(
    event.payload.orderId,
    event.payload.status
  );

  if (!validationResult.valid) {
    return {
      success: false,
      reason: validationResult.reason
    };
  }
});
```

```
    },
  }

  // Store in context
  context.orderId = event.payload.orderId;
  context.newStatus = event.payload.status;

  // Apply state transformation
  return applyStatusChange;
});

// Function that applies state transformation
function applyStatusChange(event, context) {
  const { orderId, newStatus } = context;

  // Transform state
  orderNode.state.set(state => {
    const orders = state.orders || {};
    const order = orders[orderId];

    if (!order) {
      return state;
    }

    return {
      ...state,
      orders: {
        ...orders,
        [orderId]: {
          ...order,
          status: newStatus,
          updatedAt: Date.now()
        }
      }
    };
  });

  // Emit another event based on the state change
  orderNode.broadcast({
    type: "order-status-changed",
    payload: {
      orderId,
      status: newStatus
    }
  });
});
```

```
// Return success result
return {
  success: true,
  orderId,
  status: newStatus
};
}
```

This unified approach allows you to seamlessly combine the reactive, event-driven flow with the consistent, state-focused transformations, giving you the best of both worlds.

## NATS: The Underlying Fabric

While Happen's conceptual model is built around Nodes and Events, the framework leverages NATS as its underlying messaging fabric to provide:

1. **Distributed Messaging:** High-performance communication between nodes, even across network boundaries
2. **Persistence:** Durable storage of events and state through JetStream
3. **Exactly-Once Processing:** Guaranteed message delivery and processing semantics
4. **Cross-Environment Operation:** Unified communication across server, browser, and edge environments

This foundation ensures that Happen systems are not only conceptually clean but also robust and resilient in real-world distributed environments.

Now that you understand the core concepts of Happen, you're ready to explore how these primitives combine to create powerful communication patterns and sophisticated system behaviors.

In the next section, we'll explore event pattern matching and how it enables powerful selective event handling.

# Event Pattern Matching

In Happen, we've taken a different approach to event pattern matching. Instead of providing complex pattern syntax with its own parsing rules and limitations, we leverage JavaScript's first-class functions to give you complete freedom in defining how events should be matched.

## Function-Based Matchers

At its core, a pattern in Happen is simply a function that decides whether an event should be handled:

```
// A pattern is a function that returns true or false for an event type
node.on(eventType => eventType === 'order.submitted', (event, context) => {
  // Process order submission
  // ...
  // Return next function or result
  return processPayment;
});
```

This function receives the event's type string and returns a boolean: true if the event should be handled, false if it should be ignored.

This simple approach provides extraordinary flexibility:

```
// Match exact event type
node.on(type => type === 'order.submitted', handleOrderSubmission);

// Match events by domain prefix
node.on(type => type.startsWith('order.'), (event, context) => {
  // Log all order events
  logOrderEvent(event);
  // Continue to domain-specific handler
  return getDomainSpecificHandler(event.type);
});
```

```
},

// Match multiple specific events
node.on(type => ['payment.succeeded',
'payment.failed'].includes(type),
function(event, context) {
  // Process payment result
  processPaymentResult(event);
  // Branch based on success or failure
  return event.type === 'payment.succeeded' ?
    handlePaymentSuccess : handlePaymentFailure;
});

// Match with regular expressions
node.on(type => /^user\.(created|updated|deleted)$/.test(type),
function(event, context) {
  // Handle user lifecycle event
  updateUserCache(event);
  // Determine next step based on event type
  const actions = {
    'user.created': notifyUserCreated,
    'user.updated': notifyUserUpdated,
    'user.deleted': notifyUserDeleted
  };
  // Return appropriate next function
  return actions[event.type];
});

// Complex conditional matching
node.on(type => {
  const [domain, action] = type.split('.');
  return domain === 'inventory' && action.includes('level');
}, function(event, context) {
  // Process inventory level event
  processInventoryLevel(event);
  // Check if restock needed
  if (isRestockNeeded(event.payload)) {
    return createRestockOrder;
  }
  // Otherwise complete
  return { processed: true };
});
```

## Creating Your Own Pattern System

With function-based matchers, you can build any pattern matching system that suits your needs. Here's an example of how you might create your own pattern utilities:

```
// Create a domain matcher
function domain(name) {
  return type => type.startsWith(`${name}.`);
}

// Create an exact matcher
function exact(fullType) {
  return type => type === fullType;
}

// Create a wildcard matcher
function wildcard(pattern) {
  // Convert the pattern to a regex
  const regex = new RegExp('^' + pattern.replace(/\./g, '\\.').replace(/\*/g, '.*') + '$');
  return type => regex.test(type);
}

// Create a matcher for multiple patterns
function oneOf(...patterns) {
  return type => patterns.some(pattern =>
    typeof pattern === 'function' ? pattern(type) : type ===
    pattern
  );
}

// Create a matcher that excludes patterns
function not(pattern) {
  const matcher = typeof pattern === 'function' ? pattern : type =>
    type === pattern;
  return type => !matcher(type);
}
```

Now you can use these utilities to create expressive, reusable matchers:

```

// Match all order events
node.on(domain('order'), (event, context) => {
  // Process all order events
  logOrderEvent(event);
  // Continue to specific handler based on event type
  const actionType = event.type.split('.')[1];
  return getHandlerForAction(actionType);
});

// Match a specific event exactly
node.on(exact('payment.succeeded'), (event, context) => {
  // Process successful payment
  updateOrderPaymentStatus(event.payload);
  // Continue to shipping
  return createShipment;
});

// Match using wildcards
node.on(wildcard('user.*.completed'), (event, context) => {
  // Handle user completion events
  trackUserCompletion(event);
  // Determine next steps
  return selectNextUserAction;
});

// Match any of multiple patterns
node.on(oneOf(
  exact('order.submitted'),
  exact('payment.succeeded'),
  wildcard('shipping.*')
), (event, context) => {
  // Handle important events
  notifyAdmins(event);
  // Continue to regular processing
  return processNormally;
});

// Match one domain but exclude specific events
node.on(
  type => domain('user')(type) && not(exact('user.password-
changed'))(type),
  (event, context) => {
    // Handle non-sensitive user events
    logUserAction(event);
  }
);

```

```
// Continue to specific handler
return getHandlerForUserAction(event.type);
}
);
```

## String Pattern Support

For convenience, Happen also supports string patterns which are converted to matcher functions internally:

```
// These are equivalent
node.on('order.submitted', handleOrderSubmission);
node.on(type => type === 'order.submitted', handleOrderSubmission);

// These are equivalent too
node.on('order.*', handleAllOrderEvents);
node.on(type => type.startsWith('order.') && !type.includes('.',
type.indexOf('.') + 1), handleAllOrderEvents);
```

String patterns support several features:

- Exact matching: 'order.submitted'
- Wildcards: 'order.\*'
- Alternative patterns: '{order,payment}.created'
- Multiple segments: 'user.profile.\*'

However, function matchers provide greater flexibility and expressiveness when you need more complex matching logic.

## Building Domain-Specific Pattern Systems

For larger applications, you might want to build a more structured pattern system. Here's an example of a domain-oriented approach:



```
// Create a domain builder
function createDomain(name) {
  // Basic event matcher creator
  function eventMatcher(action) {
    return type => type === `${name}.${action}`;
  }

  // Return a function that can be called directly or accessed for
  // utilities
  const domain = eventMatcher;

  // Add utility methods
  domain.all = () => type => type.startsWith(`${name}.`);
  domain.oneOf = (...actions) => type => {
    const prefix = `${name}.`;
    return type.startsWith(prefix) &&
      actions.includes(type.slice(prefix.length));
  };

  return domain;
}

// Create domains for your application
const order = createDomain('order');
const payment = createDomain('payment');
const user = createDomain('user');

// Use them in your handlers
node.on(order('submitted'), (event, context) => {
  // Handle order submission
  processOrder(event.payload);
  // Return next function
  return validateInventory;
});

node.on(order.all(), (event, context) => {
  // Log all order events
  logOrderActivity(event);
  // Continue normal processing
  return context.next || null;
});

node.on(payment.oneOf('succeeded', 'failed'), (event, context) => {
  // Process payment results
  updateOrderWithPayment(event);
});
```

```
// Branch based on result
return event.type === 'payment.succeeded' ?
  handleSuccessfulPayment : handleFailedPayment;
});
```

## Benefits of Function-Based Pattern Matching

This approach to pattern matching offers several advantages:

1. Unlimited Flexibility: Any matching logic can be implemented
2. Zero Parse-Time Overhead: Patterns are just functions, no parsing needed
3. Type Safety: TypeScript can fully type your pattern functions
4. Testability: Pattern functions can be unit tested independently
5. Composition: Combine matchers to create complex patterns
6. Familiar JavaScript: No special syntax to learn, just standard JS

## Best Practices

- Keep matcher functions pure: They should depend only on the input event type
- Create reusable pattern factories: Build a library of matcher creators for your application
- Compose simple matchers: Build complex patterns by combining simple ones
- Test your matchers: Unit test complex matching logic independently
- Consider performance: For high-frequency events, optimize your matcher functions

# The Event Continuum

In Happen, we've distilled event processing to its purest essence: a continuous flow of functions that process events and determine what happens next.

## The Pure Functional Flow Model

At its core, the Event Continuum views event handling as a single entry point that can branch into a chain of functions through return values:

```
// Register a single entry point
orderNode.on("create-order", validateOrder);

// The flow is controlled by function returns
function validateOrder(event, context) {
  if (!isValidOrder(event.payload)) {
    return { success: false, reason: "Invalid order" };
  }

  // Return the next function to execute
  return processOrder;
}

function processOrder(event, context) {
  // Process the order
  const orderId = createOrderInDatabase(event.payload);

  // Store in context
  context.orderId = orderId;

  // Return the next function
  return notifyCustomer;
}

function notifyCustomer(event, context) {
  // Send notification using context data
  sendOrderConfirmation(context.orderId, event.payload.customer);

  // Return the next function
}
```

```
// Return final result
return { success: true, orderId: context.orderId };
}
```

This creates an elegant flow system with just one fundamental mechanism: **functions that return either the next function or a final value.**

## How the Continuum Works

When an event arrives at a node, the system:

1. Calls the registered handler function
2. Examines the return value:
  - If it's a function: Execute that function next
  - If it's any other value: Complete the flow with that value as the result

This simple mechanism creates a remarkably powerful and flexible flow system.

## Flow Control Through Return Values

The Event Continuum achieves its power through a pure functional approach:

### Continuing to the Next Step

When a handler returns another function, the flow continues with that function:

```
function checkInventory(event, context) {
  // Check inventory
  const inventoryResult = validateStock(event.payload.items);

  if (!inventoryResult.available) {
    return handleInventoryShortage;
  }
}
```

```
}

// Continue to payment processing
return processPayment;
}
```

## Completing the Flow with a Result

When a handler returns any non-function value, the flow completes with that value as the result:

```
function processPayment(event, context) {
  // Process payment
  const paymentResult = chargeCustomer(event.payload.payment);

  if (!paymentResult.success) {
    // Complete with failure result
    return {
      success: false,
      reason: "payment-failed",
      details: paymentResult.error
    };
  }

  // Continue to shipping
  return createShipment;
}
```

## Dynamic Flow Selection

Functions can return different next steps based on any condition:

```
function determinePaymentMethod(event, context) {
  // Choose next step based on payment method
  const method = event.payload.paymentMethod;

  // Return different functions based on conditions
  if (method === "credit-card") {
    return processCreditCard;
  }
}
```

```
    } else if (method === "paypal") {  
      return processPaypal;  
    } else {  
      return processAlternativePayment;  
    }  
  }  
}
```

## Shared Context

A shared context object flows through the function chain, allowing communication between steps:

```
function validateOrder(event, context) {  
  // Store validation result in context  
  context.validation = validateOrderData(event.payload);  
  
  if (!context.validation.valid) {  
    return { success: false, errors: context.validation.errors };  
  }  
  
  return processInventory;  
}  
  
function processInventory(event, context) {  
  // Access context from previous step  
  const validatedItems = context.validation.items;  
  
  // Add more to context  
  context.inventory = checkInventoryForItems(validatedItems);  
  
  return processPayment;  
}
```

This provides a clean way for functions to build up state as the flow progresses.

## Complex Flow Patterns

The pure functional model supports sophisticated flow patterns:

## Conditional Branching

```
function processOrder(event, context) {  
  // Branch based on order type  
  if (event.payload.isRush) {  
    return processRushOrder;  
  } else if (event.payload.isInternational) {  
    return processInternationalOrder;  
  } else {  
    return processStandardOrder;  
  }  
}
```

## Loop Patterns

You can create loops by returning a function that's already been executed:

```
function processItems(event, context) {  
  // Initialize if first time  
  if (!context.itemIndex) {  
    context.itemIndex = 0;  
    context.results = [];  
  }  
  
  // Get current item  
  const items = event.payload.items;  
  const currentItem = items[context.itemIndex];  
  
  // Process current item  
  const result = processItem(currentItem);  
  context.results.push(result);  
  
  // Increment index  
  context.itemIndex++;  
  
  // Loop if more items  
  if (context.itemIndex < items.length) {  
    return processItems; // Loop back to this function  
  }  
}
```

```
}

// Otherwise, continue to next handler
return summarizeResults;
}
```

## Error Handling Patterns

Error handling becomes part of the natural flow:

```
function processPayment(event, context) {
  try {
    // Attempt to process payment
    const result = chargeCustomer(event.payload.payment);
    context.payment = result;

    // Continue to shipping
    return createShipment;
  } catch (error) {
    // Handle error
    context.error = error;

    // Branch to error handler
    return handlePaymentError;
  }
}

function handlePaymentError(event, context) {
  // Log error
  logPaymentFailure(context.error);

  // Return error result
  return {
    success: false,
    reason: "payment-failed",
    error: context.error.message
  };
}
```

## Building Complete Workflows



The Event Continuum naturally supports building complex workflows:

```
// Entry point
orderNode.on("process-order", validateOrder);

// Define workflow steps as functions
function validateOrder(event, context) {
  const validation = performValidation(event.payload);

  if (!validation.valid) {
    return { success: false, errors: validation.errors };
  }

  // Store in context and continue
  context.validation = validation;
  return checkInventory;
}

function checkInventory(event, context) {
  const inventoryResult = checkStockLevels(event.payload.items);

  if (!inventoryResult.available) {
    return handleInventoryShortage;
  }

  // Store in context and continue
  context.inventory = inventoryResult;
  return processPayment;
}

function processPayment(event, context) {
  const paymentResult = processTransaction(event.payload.payment);

  if (!paymentResult.success) {
    return handlePaymentFailure;
  }

  // Store in context and continue
  context.payment = paymentResult;
  return createShipment;
}

function createShipment(event, context) {
```

```
function createShipment(event, context) {  
  // Create shipment  
  const shipment = generateShipment(event.payload.address,  
    event.payload.items);  
  context.shipment = shipment;  
  
  return finalizeOrder;  
}  
  
function finalizeOrder(event, context) {  
  // Finalize the order  
  const finalOrder = {  
    orderId: generateOrderId(),  
    customer: event.payload.customer,  
    payment: context.payment,  
    shipment: context.shipment,  
    status: "completed"  
  };  
  
  // Save order  
  saveOrderToDatabase(finalOrder);  
  
  // Return final result  
  return {  
    success: true,  
    order: finalOrder  
  };  
}  
  
// Error handling functions  
function handleInventoryShortage(event, context) {  
  notifyInventoryTeam(event.payload.items);  
  
  return {  
    success: false,  
    reason: "inventory-shortage",  
    availableOn: estimateRestockDate(event.payload.items)  
  };  
}  
  
function handlePaymentFailure(event, context) {  
  // Log failure  
  logPaymentFailure(event.payload.payment);  
  
  return {  
    success: false.  
  }  
}
```

```
    reason: "payment-failed",  
    paymentError: context.payment.error  
  };  
}
```

## The Power of Pure Functional Flows

This pure functional approach offers several benefits:

1. **Ultimate Simplicity:** One consistent pattern for all event handling
2. **Maximum Flexibility:** Functions can compose and branch in unlimited ways
3. **Transparent Logic:** Flow control is explicit in function returns
4. **Perfect Testability:** Each function can be tested independently
5. **Minimal API Surface:** Just `.on()` with a single handler

This approach embodies Happen's philosophy in its purest form - a single registration method and function returns create a system of unlimited expressiveness.

## Examples of the Event Continuum in Action

### Simple Event Handling

```
// Simple handler that completes immediately  
orderNode.on("order-created", (event) => {  
  console.log("Order created:", event.payload.id);  
  return { acknowledged: true };  
});
```

### Validation Pattern

```
// Entry point with validation
orderNode.on("create-user", (event) => {
  const validation = validateUserData(event.payload);

  if (!validation.valid) {
    return { success: false, errors: validation.errors };
  }

  // Continue to user creation
  return createUserRecord;
});

function createUserRecord(event, context) {
  // Create user in database
  const userId = createUser(event.payload);

  // Return success
  return { success: true, userId };
}
```

## Request-Response with Direct Return

```
// Direct communication with explicit return
dataNode.on("get-user-data", (event, context) => {
  const { userId, fields } = event.payload;

  // Fetch user data
  const userData = fetchUserData(userId, fields);

  // Return directly to requester - this value will be sent back
  return userData;
});
```

## Ending Flow with Bare Return

A bare `return` statement will end the flow without returning any value:

```
// End flow with a bare return statement
dataNode.on("get-user-data", (event, context) => {
```

```
dataNode.on('log-activity', (event, context) => {  
  // Log the activity  
  logUserActivity(event.payload.userId, event.payload.action);  
  
  // End the flow with a bare return  
  return;  
});
```

This is equivalent to returning `undefined` but is more explicit about the intention to end the flow without a value.

## Building Reusable Flow Patterns

The functional nature of the Event Continuum encourages building reusable patterns:

```
// Create a validation wrapper  
function withValidation(validator, nextStep) {  
  return function(event, context) {  
    const validation = validator(event.payload);  
  
    if (!validation.valid) {  
      return { success: false, errors: validation.errors };  
    }  
  
    // Store validation result  
    context.validation = validation;  
  
    // Continue to next step  
    return nextStep;  
  };  
}  
  
// Create a retry wrapper  
function withRetry(handler, maxRetries = 3) {  
  return function retryHandler(event, context) {  
    // Initialize retry count  
    context.retryCount = context.retryCount || 0;  
  
    try {  
      // Attempt to execute handler
```

```

    return handler(event, context);
  } catch (error) {
    // Increment retry count
    context.retryCount++;

    // Check if we can retry
    if (context.retryCount <= maxRetries) {
      console.log(`Retrying
(${context.retryCount}/${maxRetries})...`);
      return retryHandler; // Recurse to retry
    }

    // Too many retries
    return {
      success: false,
      error: error.message,
      retries: context.retryCount
    };
  }
};

// Usage
orderNode.on("create-order",
  withValidation(validateOrder,
    withRetry(processOrder)
  )
);

```

## Parallel Processing

For more advanced scenarios, you can implement parallel processing:

```

// Parallel execution helper
function parallel(functions) {
  return async function(event, context) {
    // Execute all functions in parallel
    const results = await Promise.all(
      functions.map(fn => fn(event, context))
    );
  };
}

```

```
// Store results
context.parallelResults = results;

// Return the continuation function
return context.continuation;
};
}

// Usage
orderNode.on("process-order", (event, context) => {
  // Set up the continuation
  context.continuation = finalizeOrder;

  // Execute these functions in parallel
  return parallel([
    validateInventory,
    processPayment,
    prepareShipping
  ]);
});

function finalizeOrder(event, context) {
  // Access results from parallel execution
  const [inventoryResult, paymentResult, shippingResult] =
    context.parallelResults;

  // Proceed based on all results
  if (inventoryResult.success && paymentResult.success &&
    shippingResult.success) {
    return { success: true, order:
      combineResults(context.parallelResults) };
  } else {
    return { success: false, failures:
      findFailures(context.parallelResults) };
  }
}
```

By treating event handling as a pure functional flow where each function determines what happens next, Happen enables a system of unlimited expressiveness that can handle everything from simple events to complex workflows with the same consistent pattern.

Ready to explore more? Continue to the Communication Patterns section to

see how the Event Continuum integrates with other aspects of the Happen framework.



# Communication Patterns

The lifeblood of any Happen system is communication between nodes. Happen distills communication down to fundamental patterns that can be composed to create any interaction model.

## System-wide Broadcasting

System-wide broadcasting creates an environment where information flows freely throughout the entire ecosystem. Any node can transmit events that propagate across the system, reaching all nodes without targeting specific recipients.

```
// Weather monitoring service broadcasts a system-wide alert
weatherMonitor.broadcast({
  type: "severe-weather",
  payload: {
    condition: "hurricane",
    region: "Gulf Coast",
    expectedImpact: "severe",
    timeframe: "36 hours",
  },
});

// Any node in the system can listen for these alerts
emergencyServices.on(type => type === "severe-weather", event => {
  deployEmergencyTeams(event.payload.region);
});
```

Broadcasting is ideal for:

- System-wide notifications
- Important state changes that many nodes might care about
- Environmental changes affecting the entire system
- Crisis or exceptional condition alerts

## Direct Communication

Direct communication establishes dedicated point-to-point channels between nodes, enabling private exchanges, guaranteed delivery, and response-oriented interactions.

## Request-Response Pattern

```
// Order service directly communicates with payment service
async function processOrder(event, context) {
  // Send to payment service and await response using .return()
  const result = await orderNode.send(paymentNode, {
    type: "process-payment",
    payload: {
      orderId: event.payload.orderId,
      amount: calculateTotal(event.payload.items),
      currency: "USD",
      customerId: event.payload.customerId,
    }
  }).return();

  // Handle the response
  if (result.status === "approved") {
    return createShipment;
  } else {
    return handlePaymentFailure;
  }
}

// Payment service handles the request
paymentNode.on("process-payment", event => {
  // Process the payment
  const result = chargeCustomer(event.payload);

  // Return result directly
  return {
    status: result.success ? "approved" : "declined",
    transactionId: result.transactionId,
    message: result.message
  };
});
```

```
},
```

The `.return()` method explicitly indicates that you're expecting a response, making the code more readable and intention-clear. The receiving node simply returns a value from its handler as usual.

## Response Handling with Callbacks

For more sophisticated response handling, you can provide a callback to

`.return()`:

```
// Send request and handle response with a callback
orderNode.send(inventoryNode, {
  type: "check-inventory",
  payload: { itemId: "123", quantity: 5 }
}).return(response => {
  if (response.available) {
    processAvailableItem(response);
  } else {
    handleOutOfStock(response);
  }
});
```

## Fire-and-Forget Communication

When no response is needed, simply don't call `.return()`:

```
// Notification without needing a response
function notifyCustomer(event, context) {
  // Send notification without calling .return()
  orderNode.send(emailNode, {
    type: "send-email",
    payload: {
      to: event.payload.customerEmail,
      subject: "Order Confirmation",
      orderId: event.payload.orderId
    }
  });
}
```

```
// Continue processing immediately
return finalizeOrder;
}

// Email node doesn't need to return anything
emailNode.on("send-email", event => {
  sendCustomerEmail(event.payload);
  // No explicit return needed for fire-and-forget
});
```

This approach has several advantages:

- The sending node clearly indicates its expectations
- The API is more extensible for future enhancements
- It provides a cleaner way to handle responses
- It avoids relying on node context details

Direct communication is best for:

- Request-response interactions
- Private or sensitive information exchange
- Operations requiring acknowledgment
- Complex workflows requiring coordination

## Implicit Contracts

In Happen, contracts between nodes emerge naturally from event patterns:

- A node's interface is defined by the events it handles and emits
- The causal relationships between events form a natural contract
- New nodes can understand existing patterns through observation
- Contracts can evolve naturally as systems change

This emergent approach eliminates the need for formal interface definitions or schema registries.

## Choosing the Right Pattern

While Happen allows you to freely mix communication patterns, here are some guidelines:

- Use broadcasting when information needs to reach multiple recipients
- Use direct communication for targeted interactions
- Use `.return()` when you need a response
- Skip `.return()` for fire-and-forget operations

The right combination of patterns will depend on your specific domain and requirements. By leveraging these fundamental communication patterns and composing them in different ways, you can create a wide range of interaction models that fit your specific needs.

# Confluence

## Unified Fan-in and Fan-out Event Processing

Confluence is Happen's system for handling multiple events and multiple nodes with minimal API surface. It provides powerful capabilities for both event batching (fan-in) and multi-node distribution (fan-out) through a single, intuitive container pattern.

### Core Idea

Confluence embodies a simple concept: **When something is in a container, it represents a collection.** This principle applies consistently whether you're working with nodes or events:

- An array of nodes means "multiple receivers"
- An array of events means "a batch of events"
- Handlers naturally work with both individual items and collections

This symmetry creates a powerful system with virtually no new API surface.

### API Surface

The entire Confluence system introduces zero new methods, instead extending Happen's existing API to work with arrays:

### Fan-out: One Event, Multiple Nodes

```
// Register a handler across multiple nodes using an array
[orderNode, paymentNode, inventoryNode].on("update", (event,
context) => {
  // ...
})
```

```
// Handler receives events from any of these nodes
console.log(`Processing update in ${context.node.id}`);
});

// Send an event to multiple nodes using an array
[orderNode, shippingNode, notificationNode].send({
  type: "order-completed",
  payload: { orderId: "ORD-123" }
});
```

## Fan-in: Multiple Events, One Handler

```
// Send multiple events as a batch using an array
node.send(targetNode, [
  { type: "data-point", payload: { value: 10 } },
  { type: "data-point", payload: { value: 20 } },
  { type: "data-point", payload: { value: 30 } }
]);

// Handler naturally works with both individual events and batches
node.on("data-point", (eventOrEvents, context) => {
  // Simple array check tells you what you received
  if (Array.isArray(eventOrEvents)) {
    console.log(`Processing batch of ${eventOrEvents.length}
events`);
    return processBatch(eventOrEvents);
  } else {
    console.log(`Processing single event:
${eventOrEvents.payload.value}`);
    return processSingle(eventOrEvents);
  }
});
```

## The Event Continuum and Divergent Flows

When an event is sent to multiple nodes using Confluence, each node processes it through its own independent flow chain, creating "divergent flows" - parallel processing paths that naturally extend the Event Continuum model:

```
// Register with multiple nodes
[orderNode, inventoryNode, notificationNode].on("order-updated",
function validateUpdate(event, context) {
  // This function runs independently for each node
  console.log(`Validating update in ${context.node.id}`);

  // Each node can return a different next function
  if (context.node.id === "order-service") {
    return updateOrderRecord;
  } else if (context.node.id === "inventory-service") {
    return updateInventoryLevels;
  } else {
    return sendNotifications;
  }
});

// Node-specific continuation functions
function updateOrderRecord(event, context) {
  // Only runs in the order node's flow
  console.log("Updating order records");
  return { updated: true };
}

function updateInventoryLevels(event, context) {
  // Only runs in the inventory node's flow
  console.log("Updating inventory levels");
  return { inventoryUpdated: true };
}

function sendNotifications(event, context) {
  // Only runs in the notification node's flow
  console.log("Sending notifications");
  return { notificationsSent: true };
}
```

This creates a causal tree structure where:

- Each node has its own isolated context
- Flow paths can diverge based on node-specific logic
- Return values are tracked per node
- The complete causal history is preserved



When you need the results from multiple nodes:

```
// Send to multiple nodes and await results using the standard
return method
const results = await sender.send([node1, node2, node3],
event).return();

// Results contains node-specific returns in a keyed object
console.log(`Node1 result: ${results.node1}`);
console.log(`Node2 result: ${results.node2}`);
console.log(`Node3 result: ${results.node3}`);
```

## No Config Needed

Since batching is fully explicit with arrays, there's no magical configuration needed. Batches are simply arrays of events that you create and send directly:

```
// Send a single event
node.send(targetNode, {
  type: "data-point",
  payload: { value: 42 }
});

// Send a batch as an explicit array of events
node.send(targetNode, [
  { type: "data-point", payload: { value: 10 } },
  { type: "data-point", payload: { value: 20 } },
  { type: "data-point", payload: { value: 30 } }
]);
```

This direct approach ensures complete predictability with no behind-the-scenes magic.

## Causality Preservation

Even with batches and multi-node processing, Confluence maintains Happen's causality guarantees:

- The causal context is preserved for each event in a batch
- Each event maintains its position in the causal chain
- Batch processing still records each event's causal history
- Divergent flows are tracked as branches in the causal tree

This means you can always trace the complete history of events, even when they've been processed in batches or across multiple nodes.

## Context in Multi-Node Operations

When working with multiple nodes, context handling is structured to maintain node-specific isolation while providing clear access to relevant information:

### Node-Specific Context Structure

When a handler receives an event in a multi-node operation, the context structure clearly identifies which node is processing it:

```
[orderNode, inventoryNode, shippingNode].on("order-updated",
(event, context) => {
  // The context.node property identifies the specific node
  console.log(`Processing in: ${context.node.id}`);

  // Each node maintains its own isolated context
  context.nodeState = context.nodeState || {};
  context.nodeState.lastProcessed = Date.now();
});
```

## Results Collection

When collecting results from multiple nodes, the returned object has a clear

structure with node IDs as keys:

```
// Send to multiple nodes and collect results
const results = await sender.send([orderNode, inventoryNode,
shippingNode], {
  type: "verify-order",
  payload: { orderId: "ORD-123" }
}).return();

// Results object is structured by node ID
console.log(`Order verification:
${results[orderNode.id].verified}`);
console.log(`Inventory status:
${results[inventoryNode.id].available}`);
console.log(`Shipping estimate:
${results[shippingNode.id].estimatedDays}`);
```

## Batch Context Structure

For batch operations (multiple events to a single node), the context provides batch-level information at the root, with individual event contexts in an array:

```
node.on("data-point", (events, context) => {
  if (Array.isArray(events)) {
    // Batch-level information at the root
    console.log(`Processing batch of ${events.length} events`);
    console.log(`Batch received at: ${context.receivedAt}`);

    // Each event has its own context in the events array
    events.forEach((event, index) => {
      // context.events contains the individual event contexts
      const eventContext = context.events[index];
      console.log(`Event ${index} causation:
${eventContext.causal.causationId}`);
    });
  }
});
```

## Combined Multi-Node and Batch Operations

In the rare case of both batching and multi-node operations, the context structure maintains clear separation:

```
[nodeA, nodeB].on("process-batch", (eventsOrEvent, context) => {
  // First determine which node we are
  const nodeId = context.node.id;

  // Then check if we're processing a batch
  if (Array.isArray(eventsOrEvent)) {
    // We're in a specific node, processing a batch
    console.log(`Node ${nodeId} processing batch of
    ${eventsOrEvent.length} events`);

    // Access node-specific batch processing logic
    return
    nodeProcessingStrategies[nodeId].processBatch(eventsOrEvent,
    context);
  } else {
    // We're in a specific node, processing a single event
    console.log(`Node ${nodeId} processing single event`);

    // Access node-specific individual processing logic
    return
    nodeProcessingStrategies[nodeId].processSingle(eventsOrEvent,
    context);
  }
});
```

This context structure ensures that no matter how complex the operation, each node maintains its own isolated processing environment while still providing clear access to all necessary information.

## Examples

### Explicit Batch Creation and Processing

```
// Efficiently handle high-volume sensor data
const telemetryNode = createNode("telemetry-processor");
```

```

// Explicitly create batches - no magic behind the scenes
function sendSensorData(readings) {
  // Create batches of appropriate size
  const batchSize = 50;

  for (let i = 0; i < readings.length; i += batchSize) {
    // Explicitly create a batch as an array of events
    const batch = readings.slice(i, i + batchSize).map(reading =>
    ({
      type: "sensor-reading",
      payload: {
        sensorId: reading.sensorId,
        value: reading.value
      }
    }));

    // Send the batch explicitly as an array
    sensorNode.send(telemetryNode, batch);
  }
}

// Generate some sample readings
const readings = Array.from({ length: 1000 }, (_, i) => ({
  sensorId: `sensor-${i % 10}`,
  value: Math.random() * 100
}));

// Send readings in explicit batches
sendSensorData(readings);

// Handler processes either single events or explicitly sent
batches
telemetryNode.on("sensor-reading", (eventOrEvents, context) => {
  if (Array.isArray(eventOrEvents)) {
    // Process an explicitly sent batch
    const readings = eventOrEvents.map(e => e.payload.value);
    const average = readings.reduce((sum, v) => sum + v, 0) /
readings.length;

    console.log(`Processed batch of ${eventOrEvents.length}
readings, avg: ${average}`);
    return { processed: readings.length };
  } else {
    // Process single event
    // ... Single Reading (eventOrEvents is a single event)

```

```

    processSingleReading(eventOrEvents.payload);
    return { processed: 1 };
  }
});

```

## Distributed Notification System with Divergent Flows

```

// System monitor that alerts multiple subsystems
const monitorNode = createNode("system-monitor");
const alertNode = createNode("alert-service");
const loggingNode = createNode("logging-service");
const adminNode = createNode("admin-dashboard");

// Send critical alerts to multiple services
function reportCriticalIssue(issue) {
  [alertNode, loggingNode, adminNode].send({
    type: "critical-alert",
    payload: {
      issue,
      timestamp: Date.now(),
      severity: "critical"
    }
  });
}

// Each node processes alerts through its own flow
[alertNode, loggingNode, adminNode].on("critical-alert", function
processAlert(event, context) {
  // Common processing logic
  console.log(`Alert received in ${context.node.id}`);

  // Node-specific next step
  if (context.node.id === "alert-service") {
    return sendUserAlerts;
  } else if (context.node.id === "logging-service") {
    return createLogEntry;
  } else {
    return updateDashboard;
  }
});

// Node-specific continuation functions
function sendUserAlerts(event, context) {

```

```
// Alert service sends notifications to users
return { alertsSent: 10 };
}

function createLogEntry(event, context) {
  // Logging service creates permanent record
  return { loggedAt: Date.now() };
}

function updateDashboard(event, context) {
  // Admin dashboard updates UI
  return { dashboardUpdated: true };
}
```

## Manual Batch Processing

```
// Explicitly create and send a batch of events
const orderBatch = [
  { type: "process-order", payload: { orderId: "ORD-001", total:
125.00 } },
  { type: "process-order", payload: { orderId: "ORD-002", total:
75.50 } },
  { type: "process-order", payload: { orderId: "ORD-003", total:
240.00 } }
];

// Send the batch
node.send(processingNode, orderBatch);

// Handle the batch on the receiving side
processingNode.on("process-order", (eventOrEvents, context) => {
  if (Array.isArray(eventOrEvents)) {
    // Batch processing
    const totalValue = eventOrEvents.reduce((sum, e) => sum +
e.payload.total, 0);
    return { totalProcessed: eventOrEvents.length, totalValue };
  } else {
    // Individual processing
    return { processed: eventOrEvents.payload.orderId };
  }
});
```

# Performance Considerations

Confluence is designed to be explicit and predictable while still providing performance benefits:

## When to Use Batching

- High-volume event streams with similar event types
- Processing that benefits from aggregation (like analytics)
- Networks with significant latency where reducing round-trips helps
- Calculations that can be optimized when performed on multiple items together

## When to Use Multi-node Operations

- Broadcast notifications that multiple subsystems need to process
- Commands that affect multiple services simultaneously
- Cross-cutting concerns like logging, monitoring, or security
- Redundant processing for critical operations

## Batch Processing Efficiency

For maximum efficiency when processing batches:

1. **Use Specialized Algorithms:** Many operations are more efficient on batches (like bulk database inserts)
2. **Minimize Per-event Overhead:** Amortize setup costs across multiple events
3. **Leverage Memory Locality:** Process related data together to improve cache efficiency



4. **Prefer Single Passes:** Process the entire batch in one pass rather than multiple iterations

## Memory Management

Batch processing can also help with memory efficiency:

```
// Process high-volume data with controlled memory usage
node.on("high-volume-data", (eventOrEvents, context) => {
  if (Array.isArray(eventOrEvents)) {
    // Process incrementally to avoid memory spikes
    let results = [];
    const batchSize = 100;

    for (let i = 0; i < eventOrEvents.length; i += batchSize) {
      const chunk = eventOrEvents.slice(i, i + batchSize);
      const chunkResults = processChunk(chunk);
      results = results.concat(chunkResults);

      // Allow event loop to run between chunks if needed
      if (i + batchSize < eventOrEvents.length) {
        yield { progress: Math.round((i + batchSize) /
eventOrEvents.length * 100) };
      }
    }

    return { results };
  }
});
```

## Conclusion

Confluence provides powerful capabilities for handling multiple events and multiple nodes while maintaining Happen's commitment to radical simplicity. Through a single intuitive container pattern - using arrays for both nodes and events - it enables sophisticated batch processing and multi-node communication without introducing special methods or complex APIs.

The system offers:

1. **Pure Symmetry:** The same pattern works for both nodes and events
2. **Explicit Control:** No "magic" - batching and multi-node operations are explicit
3. **Divergent Flows:** Natural extension of the Event Continuum to parallel processing
4. **Zero New Methods:** Works entirely through existing APIs with array support
5. **Powerful Capabilities:** Enables sophisticated patterns with minimal complexity

Staying true to Happen's core philosophy of simplicity by recognizing that arrays naturally represent collections enables a powerful system with virtually no learning curve.

# Dependency Management

In line with Happen's philosophy of simplicity, our approach to dependencies emphasizes direct runtime access with minimal framework abstractions.

## Core Principles

Happen's dependency management follows three key principles:

1. **Runtime Transparency:** Your code has direct access to the runtime environment
2. **Minimal System Dependencies:** Only essential framework dependencies (primarily NATS) are injected at initialization
3. **Event-Based Communication:** Nodes interact through events, not traditional dependency injection

## System-Level Dependencies

System dependencies are configured once during framework initialization:

```
// Initialize Happen with system-level dependencies
const happen = initializeHappen({
  // NATS configuration as the primary dependency
  nats: {
    // Direct NATS client for server environments
    server: {
      servers: ['nats://localhost:4222'],
      jetstream: true
    },
    // WebSocket client for browser environments
    browser: {
      servers: ['wss://localhost:8443'],
      jetstream: true
    }
  },
});
```

```
// Optional: Override crypto implementation
crypto: cryptoImplementation
});

// The initialized framework provides the node creation function
const { createNode } = happen;
```

These system dependencies represent the minimal set required for Happen to function across different runtime environments. By injecting them at initialization, we maintain runtime agnosticism while ensuring consistent behavior.

## Direct Runtime Access

Rather than exposing a subset of features through abstraction layers, Happen encourages direct access to the runtime:

```
// Import runtime capabilities directly
import { WebSocketServer } from 'ws';
import * as fs from 'fs';

const notificationNode = createNode('notification-service');

// Set up WebSocket server directly using runtime capabilities
const wss = new WebSocketServer({ port: 8080 });
const clients = new Map();

// Use runtime WebSocket events directly
wss.on('connection', (ws, req) => {
  const userId = getUserIdFromRequest(req);
  clients.set(userId, ws);

  ws.on('close', () => {
    clients.delete(userId);
  });
});

// Process events using the Event Continuum
notificationNode.on('notify-user', function notifyUser(event,
```

```
context) {
  const { userId, message } = event.payload;

  // Direct runtime access - no framework abstraction
  const clientSocket = clients.get(userId);
  if (!clientSocket || clientSocket.readyState !== WebSocket.OPEN)
  {
    return {
      success: false,
      reason: "user-not-connected"
    };
  }

  // Send notification
  clientSocket.send(JSON.stringify({
    type: 'notification',
    message
  }));

  return {
    success: true,
    delivered: true
  };
};
```

This approach provides several benefits:

- **Zero overhead:** No performance penalty for accessing runtime features
- **Full capabilities:** Access to everything the runtime offers, not just what the framework exposes
- **Runtime evolution:** Immediate access to new runtime features without framework updates
- **Ecosystem compatibility:** Works seamlessly with the broader ecosystem

## Framework Agnosticism for Third-Party Libraries

When you need external libraries, import and use them directly:

```
// Import NATS client directly - no framework-specific imports
```

```
import { connect } from 'nats';

const databaseNode = createNode('database-service');

// Use the NATS client directly
const nc = await connect({ servers: 'nats://localhost:4222' });
const js = nc.jetstream();

databaseNode.on('query-users', function queryUsers(event, context)
{
  const { searchTerm } = event.payload;

  // Direct use of NATS client with KV store
  const kv = js.keyValue("users");
  const values = [];

  for await (const k of kv.keys()) {
    if (k.includes(searchTerm)) {
      const entry = await kv.get(k);
      values.push(JSON.parse(entry.string()));
    }
  }

  return {
    success: true,
    results: values
  };
});
```

## Dependency Injection Through Closures

Use closures to create handlers with access to dependencies:

```
// Initialize node with dependencies
function initializeDataNode() {
  // Create NATS client and JetStream KV store
  const nc = await connect({ servers: 'nats://localhost:4222' });
  const js = nc.jetstream();
  const kv = js.keyValue("data");

  // Create node
```

```
const dataNode = createNode('data-service');

// Register handler using closure to capture dependencies
dataNode.on('query-data', (event, context) => {
  // Use kv from closure scope
  return processQuery(kv, event, context);
});

return dataNode;
}

// Function that uses dependency passed via parameters
function processQuery(kv, event, context) {
  const { query, params } = event.payload;

  // Use dependency from parameters
  return new Promise((resolve, reject) => {
    // Use NATS KV store for data access
    try {
      const results = [];
      for (const key of query.keys) {
        const entry = await kv.get(key);
        if (entry) {
          results.push(JSON.parse(entry.string()));
        }
      }
      resolve({
        success: true,
        results
      });
    } catch (err) {
      reject(err);
    }
  });
}

// Create node
const dataNode = initializeDataNode();
```

## Flow State Through Closures

Although Happen provides a common object to share data across each flow

you can also use closures to manage both dependencies and flow state, with explicit control over the flow through function returns:

```
function initializeOrderNode() {
  // Create dependencies including NATS client
  const nc = await connect({ servers: 'nats://localhost:4222' });
  const js = nc.jetstream();
  const orderKV = js.keyValue("orders");
  const emailService = createEmailService();

  // Create node
  const orderNode = createNode('order-service');

  // Register initial handler with dependencies captured in closure
  orderNode.on('process-order', (event, context) => {
    // Create initial flow state
    const flowState = {
      orderId: generateOrderId(),
      items: event.payload.items,
      customer: event.payload.customer
    };

    // Return the first step function with dependencies and state
    in closure
    // This function will be executed next in the flow
    return validateOrder(orderKV, emailService, flowState);
  });

  return orderNode;
}

// Each flow function creates and returns the next handler in the chain
function validateOrder(orderKV, emailService, state) {
  // This function is returned by the initial handler and executed
  by the framework
  return (event, context) => {
    // Validate order
    const validationResult = validateOrderData(state.items);

    if (!validationResult.valid) {
      // Return a value (not a function) to complete the flow with
      error
      return 5
    }
  }
}
```



```

    return {
      success: false,
      errors: validationResult.errors
    };
  }

  // Update state (immutably)
  const updatedState = {
    ...state,
    validated: true,
    validatedAt: Date.now()
  };

  // Return the next function to be executed in the flow
  return processPayment(orderKV, emailService, updatedState);
};
}

function processPayment(orderKV, emailService, state) {
  // Return a function that will be executed as the next step
  return (event, context) => {
    // Process payment asynchronously
    const paymentResult = processCustomerPayment(state.customer,
      calculateTotal(state.items));

    if (!paymentResult.success) {
      // Return a value (not a function) to complete the flow with
      error
      return {
        success: false,
        reason: "payment-failed"
      };
    }

    // Update state (immutably)
    const updatedState = {
      ...state,
      paymentId: paymentResult.transactionId,
      paymentMethod: paymentResult.method,
      paidAt: Date.now()
    };

    // Return the next function to be executed in the flow
    return finalizeOrder(orderKV, emailService, updatedState);
  };
}

```

```
function finalizeOrder(orderKV, emailService, state) {  
  // Return a function that will be executed as the next step  
  return async (event, context) => {  
    // Create order in KV store  
    await orderKV.put(state.orderId, JSON.stringify({  
      id: state.orderId,  
      customer: state.customer,  
      items: state.items,  
      payment: {  
        id: state.paymentId,  
        method: state.paymentMethod  
      }  
    }));  
  
    // Send confirmation email  
    emailService.sendOrderConfirmation(  
      state.customer.email,  
      state.orderId,  
      state.items  
    );  
  
    // Return a value (not a function) to complete the flow with  
    success  
    return {  
      success: true,  
      orderId: state.orderId,  
      transactionId: state.paymentId  
    };  
  };  
};  
}
```

## How Flow Control Works With Closures

In this pattern, the flow is controlled through a clear mechanism:

1. **Each flow step returns a function:** When a handler wants to continue the flow, it returns a function that becomes the next handler.

2. **Flow continues while functions are returned:** The framework executes each returned function in sequence, as long as functions are being returned.
3. **Flow ends when a non-function is returned:** When a handler returns anything other than a function (like an object), the flow completes with that value as the result.
4. **Dependencies and state flow through function parameters:** Each step receives dependencies and state through its parameters, not through context.
5. **Each step creates the next step with updated state:** Functions create and return the next function in the chain, passing updated state and dependencies.

## Benefits of Closure-Based Dependency Management

This closure-based approach offers several advantages:

1. **Pure Functions:** Flow handlers become pure functions with explicit dependencies
2. **Clear Data Flow:** Dependencies and state are explicitly passed between functions
3. **Testability:** Each flow step can be easily tested in isolation with mocked dependencies
4. **Immutability:** Encourages immutable state updates through function parameters
5. **Separation of Concerns:** Clearly separates framework concerns from application logic

## NATS as the Primary Dependency

In the Happen framework, NATS serves as the primary dependency, providing

core messaging, persistence, and coordination capabilities. This focused approach means:

1. **Single Core Dependency:** NATS is the primary external dependency you need to understand
2. **Mature Ecosystem:** NATS has clients for all major languages and platforms
3. **Unified Capability Set:** One system provides messaging, persistence, and coordination
4. **Direct NATS Access:** Your code can access NATS capabilities directly when needed

## Dependency Management Philosophy

Happen's approach to dependencies reflects its broader philosophy: provide just enough framework to enable powerful capabilities, while getting out of the way and letting your code work directly with the runtime.

By limiting injected dependencies to only essential system needs (primarily NATS) and embracing direct runtime access and closure-based dependency management, Happen reduces complexity while maximizing flexibility.

There's no complex dependency injection system because, in most cases, you simply use JavaScript's natural closure mechanism to capture and pass dependencies where needed.

# Closures, A Primer

Closures are a fundamental concept in JavaScript that provide an elegant solution for managing state and dependencies in event-driven systems like Happen. This primer will help you understand what closures are, how they work, and how they enable powerful patterns in your applications.

## What is a Closure?

A closure is a function that "remembers" the environment in which it was created. More specifically, a closure is formed when a function retains access to variables from its outer (enclosing) scope, even after that outer function has completed execution.

```
function createGreeter(greeting) {  
  // The inner function is a closure that "captures" the greeting  
  variable  
  return function(name) {  
    return `${greeting}, ${name}!`;  
  };  
}  
  
// Create closures with different captured values  
const sayHello = createGreeter("Hello");  
const sayHi = createGreeter("Hi");  
  
// Use the closures  
console.log(sayHello("Alice")); // "Hello, Alice!"  
console.log(sayHi("Bob"));      // "Hi, Bob!"
```

In this example, `createGreeter` returns a function that "closes over" the `greeting` parameter. Each returned function remembers its own specific greeting, even after `createGreeter` has finished executing.

## How Closures Work

To understand closures, you need to understand two key JavaScript concepts: lexical scope and the function execution context.

## Lexical Scope

JavaScript uses lexical scoping, which means that functions are executed in the scope where they were defined, not where they are called:

```
function outer() {  
  const message = "Hello from outer scope!";  
  
  function inner() {  
    console.log(message); // Accesses variable from outer scope  
  }  
  
  inner();  
}  
  
outer(); // "Hello from outer scope!"
```

## Execution Context and Environment

When a function is created, it stores a reference to its lexical environment—the set of variables and their values that were in scope when the function was created. This environment reference stays with the function, even if the function is returned or passed elsewhere.

```
function createCounter() {  
  let count = 0; // Private state variable  
  
  return {  
    increment: function() {  
      count++; // Accesses the count variable from the outer scope  
      return count;  
    },  
    decrement: function() {  
      count--; // Also accesses the same count variable  
      return count;  
    }  
  };  
}
```

```
    },
    getCount: function() {
        return count;
    }
};

const counter = createCounter();
console.log(counter.increment()); // 1
console.log(counter.increment()); // 2
console.log(counter.decrement()); // 1
console.log(counter.getCount());  // 1
```

In this example, all three functions share access to the same `count` variable, creating a private state that can't be accessed directly from outside.

## Memory Management and Garbage Collection

Understanding how closures affect memory management is important for building efficient applications:

1. **Retained References:** When a function forms a closure, the JavaScript engine keeps all captured variables in memory as long as the function itself is reachable.
2. **Selective Retention:** The engine is smart enough to only retain variables that are actually referenced in the closure, not the entire scope.
3. **Potential Memory Leaks:** Closures can lead to memory leaks if you inadvertently keep references to large objects that are no longer needed.
4. **Automatic Cleanup:** When no references to a closure remain, both the closure and its environment will be garbage collected.

```
function potentialLeak() {
    // Large data structure
    const hugeData = new Array(1000000).fill("data");

    // This closure references hugeData
    function processingFunction() {
```

```
    return hugeData.length;
  }

  // This closure doesn't reference hugeData
  function safeFunction() {
    return "Safe result";
  }

  // processingFunction retains hugeData in memory
  // safeFunction doesn't retain hugeData

  return {
    withReference: processingFunction,
    withoutReference: safeFunction
  };
}

const result = potentialLeak();
// hugeData is still in memory because processingFunction
references it
```

To avoid memory leaks, it's good practice to:

- Only capture what you need in closures
- Set captured references to null when you're done with them
- Be mindful of large objects in closure scope

## Practical Uses of Closures

Closures enable several powerful programming patterns:

### 1. Data Encapsulation and Privacy

Closures provide a way to create private variables that can't be accessed directly from outside:

```
function createUser(name, initialBalance) {
  // Private variables
```



```
// -----
let userName = name;
let balance = initialBalance;

return {
  getName: () => userName,
  getBalance: () => balance,
  deposit: amount => {
    balance += amount;
    return balance;
  },
  withdraw: amount => {
    if (amount <= balance) {
      balance -= amount;
      return balance;
    }
    return "Insufficient funds";
  }
};
}

const user = createUser("Alice", 100);
console.log(user.getBalance()); // 100
user.deposit(50);
console.log(user.getBalance()); // 150
// Cannot access balance directly
console.log(user.balance);      // undefined
```

## 2. Function Factories

Closures allow you to create specialized functions based on parameters:

```
function createValidator(validationFn, errorMessage) {
  return function(value) {
    if (!validationFn(value)) {
      return { valid: false, error: errorMessage };
    }
    return { valid: true };
  };
}

const validateEmail = createValidator(
  email => /^[^@]+@[^@]+\.[^@]+$/.test(email),
```

```
    "Invalid email format"
  );

  const validatePassword = createValidator(
    password => password.length >= 8,
    "Password must be at least 8 characters"
  );

  console.log(validateEmail("user@example.com")); // { valid: true }
  console.log(validatePassword("123"));           // { valid: false,
  error: "Password must be at least 8 characters" }
```

### 3. Maintaining State in Async Operations

Closures are invaluable for preserving state across asynchronous operations:

```
function processUserData(userId) {
  // State captured in closure
  const processingStart = Date.now();
  const metrics = { steps: 0 };

  // Get user data
  fetchUser(userId).then(userData => {
    metrics.steps++;

    // Process user preferences
    return
    fetchPreferences(userData.preferencesId).then(preferences => {
      metrics.steps++;

      // Combine data
      const result = {
        user: userData,
        preferences: preferences,
        processingTime: Date.now() - processingStart,
        processingSteps: metrics.steps
      };

      // All async operations have access to the same captured
      state
      displayResults(result);
    });
  });
}
```

```
    });  
  }  
}
```

## 4. Event Handlers with Preset Data

Closures are perfect for creating event handlers that include specific data:

```
function setupButtonHandlers(buttons) {  
  buttons.forEach(button => {  
    // Each handler is a closure with its own reference to the  
    specific button  
    button.addEventListener('click', () => {  
      console.log(`Button ${button.id} was clicked`);  
      processButtonAction(button.dataset.action);  
    });  
  });  
}
```

## Closures in Happen's Event Continuum

In Happen, closures provide an elegant solution for managing dependencies and state across event flows:

```
function setupOrderProcessing(orderRepository, emailService) {  
  const orderNode = createNode('order-service');  
  
  orderNode.on('process-order', event => {  
    // Initialize flow state  
    const state = {  
      orderId: generateOrderId(),  
      items: event.payload.items,  
      customer: event.payload.customer  
    };  
  
    // Return first step with dependencies and state in closure  
    return validateOrder(orderRepository, emailService, state);  
  });  
  
  return orderNode;  
}
```

```
return validateOrder,
}

function validateOrder(repository, emailService, state) {
  // Return a handler function that has captured dependencies and
  state
  return (event, context) => {
    // Validate using captured state
    const validationResult = validateItems(state.items);

    if (!validationResult.valid) {
      return { success: false, errors: validationResult.errors };
    }

    // Update state immutably
    const updatedState = {
      ...state,
      validatedAt: Date.now()
    };

    // Return next step with updated state
    return processPayment(repository, emailService, updatedState);
  };
}
```

This pattern provides several benefits:

1. **Explicit Dependency Injection:** Dependencies are passed explicitly rather than through global state
2. **Immutable State Flow:** State changes are explicit and traceable
3. **Testable Units:** Each step can be tested independently with mocked dependencies
4. **Freedom from Context:** No need to rely on the event context for state

## Best Practices for Closures

To use closures effectively:

1. **Keep closures focused:** Capture only what you need to minimize memory usage.
2. **Use immutable patterns:** Update state by creating new objects rather than mutating existing ones.
3. **Be mindful of `this`:** Arrow functions capture the lexical `this`, while regular functions have their own `this` context.
4. **Watch for circular references:** These can prevent garbage collection.
5. **Prefer pure functions:** Closures that don't modify external state are easier to reason about.
6. **Consider performance:** For extremely hot code paths, be aware that closures have a small overhead compared to direct function calls.

By understanding and leveraging closures effectively, you can create elegant, maintainable code that naturally manages dependencies and state throughout your Happen applications.

# The Information System: State, Context, and Views

In Happen, information flows naturally through the system in multiple complementary forms. The Information System builds on our core primitives—Nodes and Events—while providing powerful capabilities for managing, accessing, and transforming information across your application. This document explains the conceptual model and practical implementation of state, context, and views in Happen.

---

The Information System is built on three fundamental concepts:

## State: What You Own

State is information that a node has authority over. It's the internal data that the node manages, updates, and is responsible for. State represents "what you know and control" within your domain.

State in Happen is:

- **Authoritative** - The owning node is the single source of truth
- **Persisted** - State can be saved and restored
- **Encapsulated** - Direct modification is restricted to the owning node
- **Verifiable** - State changes are tracked in the causal chain

```
// Basic state access
const orderNode = createNode("order-service");

// Read state directly
const state = orderNode.state.get();
const pendingOrders = Object.values(state.orders || {}).
  .filter(order => order.status === "pending");

// Transform state using a function approach
```

```
// Transition state using a function approach
orderNode.state.set(state => {
  return {
    ...state,
    orders: {
      ...state.orders,
      "order-123": {
        customerId: "cust-456",
        items: [{productId: "prod-789", quantity: 2}],
        status: "pending"
      }
    }
  };
});
```

State can also be accessed selectively, using transformations to extract only the required data:

```
// Focused state access
const activeOrderCount = orderNode.state.get(state =>
  Object.values(state.orders || {})
    .filter(order => order.status === "active")
    .length
);
```

## Global State Implementation

Happen's provides system-wide, crossboundary global state for universal data access throughout the unified event space. Global state is implemented using NATS Key-Value store and is accessible from all environments:

1. **Unified Access:** All nodes use the same `node.global` API regardless of environment

2. **Native Transport Adapters:** NATS provides native support for various transports:
  - TCP for server-to-server communication
  - WebSocket for browser and edge clients
  - TLS/WSS for secure communication
3. **Consistent Experience:** The same operations work identically across all environments

This approach leverages NATS' built-in transport capabilities to provide a seamless global state experience without requiring custom protocol adaptations.

For example, storing and retrieving data works the same way everywhere:

```
// Store data in global state
await node.global.set('user:profile:123', userProfile);

// Retrieve data from global state
const profile = await node.global.get('user:profile:123');
```

## Context: What Happened and Why

Context is information that flows with events, providing essential metadata about causality, origin, and relationships. Context represents the "why" and "how" of information flow in the system.

Context in Happen is:

- **Causal** - Tracks relationships between events
- **Automatic** - Flows naturally with events
- **Enriched** - Gains additional information as it flows
- **Structured** - Organized into specific categories

```
// Event with context
```



```
// Event with context
orderNode.broadcast({
  type: "order-created",
  payload: {
    orderId: "ORD-123",
    items: [{ productId: "PROD-456", quantity: 2 }]
  }
// Context is automatically managed by the framework
// context: {
//   causal: {
//     id: "evt-789", // Unique identifier
//     sender: "order-service", // Origin node
//     causationId: "evt-456", // Direct cause
//     correlationId: "order-123", // Transaction identifier
//     path: ["customer-node", "order-service"] // Event path
//   },
//   system: {
//     environment: "production",
//     region: "us-west"
//   },
//   user: {
//     id: "user-123",
//     permissions: ["create-order"]
//   }
// }
});
```

Context is system-managed and automatically flows with events, requiring no manual manipulation.

## Views: Windows Into Other Nodes

Views in Happen provide a window into other nodes' state, enabling coordinated operations across node boundaries. While conceptually simple, views are implemented using an elegant recursive traversal approach that leverages JavaScript's reference passing.

### The Recursive Reference Collection Mechanism

At the core of Happen's view implementation is a recursive traversal of the

node graph, combined with a shared reference mechanism.

When a node accesses a view, the system:

1. Creates a shared reference container (typically an object or array)
2. Passes this container to the target node
3. The target node deposits its state into the container
4. The original node can then access this state through the container

This mechanism provides several key advantages:

- **Performance:** No need to copy large state objects
- **Freshness:** Always gets the latest state when accessed
- **Simplicity:** Minimal API surface for powerful functionality
- **Consistency:** Predictable access patterns across different node types

## Basic View Usage

```
// Transform state with views into other nodes' state
orderNode.state.set((state, views) => {
  // Access customer data through views
  const customerData = views.customer.get(state =>
    state.customers[order.customerId]
  );

  // Access inventory data through views
  const inventoryData = views.inventory.get(state =>
    state.products[order.productId]
  );

  // Return updated state that incorporates external data
  return {
    ...state,
    orders: {
      ...state.orders,
      [order.id]: {
        ...order,
        canShip: inventoryData.inStock,
      },
    },
  };
});
```

```

        shippingAddress: customerData.address
    }
}
};
});

```

## Enhanced View Collection

For more efficient collection of state from multiple nodes in a single operation, Happen provides an enhanced collection capability:

```


orderNode.state.set((state, views) => {
    // Collect state from multiple nodes in a single operation
    const data = views.collect({
        customer: state => ({
            name: state.customers[order.customerId].name,
            address: state.customers[order.customerId].address
        }),
        inventory: state => ({
            inStock: state.products[order.productId].quantity > 0,
            leadTime: state.products[order.productId].leadTime
        }),
        shipping: state => ({
            rates: state.ratesByRegion[customerRegion],
            methods: state.availableMethods
        })
    });

    // Use the collected data
    return {
        ...state,
        orders: {
            ...state.orders,
            [order.id]: {
                ...order,
                fulfillmentPlan: createFulfillmentPlan(data)
            }
        }
    };
});

```

This approach:

- Traverses the node graph only once
- Collects specific slices of state from each relevant node
- Transforms the state during collection
- Returns a unified object with all the collected data

For more advanced view patterns and usage, see the dedicated [State Management](#)  page.

## The Unified Information System

These three concepts—State, Context, and Views—form a unified information model:

1. State represents what a node owns and controls
2. Context represents the causal flow of information through events
3. Views represent windows into state owned by others

Each serves a distinct purpose in the system, creating a complete information model without redundancy or overlap.

## How These Systems Work Together

The power of Happen's information system comes from how these complementary systems work together:

### Context + Views

Context and views work together to provide a complete picture of system behavior:

```

// Both context and views working together
orderNode.on("payment-received", event => {
  const { orderId, amount } = event.payload;
  const { correlationId } = event.context.causal;

  // Update order with view-aware state transformation
  orderNode.state.set((state, views) => {
    // Get the current order
    const currentOrder = state.orders[orderId];

    // Collect payment information through enhanced views
    const paymentData = views.collect({
      payments: state => ({
        details: state[event.payload.paymentId],
        methods: state.methods
      })
    });

    // Return updated state
    return {
      ...state,
      orders: {
        ...state.orders,
        [orderId]: {
          ...currentOrder,
          status: "paid",
          paymentId: event.payload.paymentId,
          paymentMethod: paymentData.payments.details.method,
          transactionId:
            paymentData.payments.details.transactionId,
          correlationId: correlationId, // From context
          paymentTimestamp: Date.now()
        }
      }
    };
  });

  // Emit event with proper context
  orderNode.broadcast({
    type: "order-paid",
    payload: {
      orderId,
      amount
    }
  });
  // Context is automatically managed by the framework

```

```
// context is automatically managed by the framework
});
});
```

## State + Context = Causal State History

The combination of state and context enables rich causal state history:

```
// State history using context
function getStateHistory(nodeId, statePath) {
  // Get all events that affected this state path
  const events = eventStore.getEventsForPath(nodeId, statePath);

  // Sort by causal order using context information
  const sortedEvents = sortByCausalOrder(events);

  // Reconstruct state history
  let state = undefined;
  const history = [];

  for (const event of sortedEvents) {
    state = applyEvent(state, event);
    history.push({
      state: { ...state },
      event: event.type,
      timestamp: event.metadata.timestamp,
      causedBy: event.context.causal.causationId,
      by: event.context.causal.sender
    });
  }

  return history;
}
```

## State + Views = System-Wide View

The combination of state and views provides a system-wide view:

```
// System-wide view using state and views
dashboardNode.on("generate-dashboard", event => {
```

```

// Use view-aware state transformation to gather system-wide data
dashboardNode.state.set((state, views) => {
  // Collect data from various services through enhanced views
  const systemData = views.collect({
    orders: state => ({
      pending: Object.values(state).filter(o => o.status ===
"pending").length,
      completed: Object.values(state).filter(o => o.status ===
"completed").length,
      failed: Object.values(state).filter(o => o.status ===
"failed").length
    }),
    inventory: state => ({
      lowStock: Object.values(state.products).filter(p => p.stock
< 10).length,
      outOfStock: Object.values(state.products).filter(p =>
p.stock === 0).length
    }),
    customers: state => ({
      active: Object.values(state).filter(c => c.active).length,
      new: Object.values(state).filter(c => {
        const oneDayAgo = Date.now() - (24 * 60 * 60 * 1000);
        return c.createdAt >= oneDayAgo;
      }).length
    }),
    payments: state => ({
      daily: Object.values(state)
        .filter(p => p.timestamp >= Date.now() - (24 * 60 * 60 *
1000))
        .reduce((sum, p) => sum + p.amount, 0),
      weekly: Object.values(state)
        .filter(p => p.timestamp >= Date.now() - (7 * 24 * 60 *
60 * 1000))
        .reduce((sum, p) => sum + p.amount, 0)
    })
  });

  // Return updated dashboard state
  return {
    ...state,
    dashboard: systemData
  };
});

```

## Practical Example: Coordinated Operations

Here's a complete example showing how views enable coordinated operations across node boundaries:

```
// Order fulfillment process using views
fulfillmentNode.on('fulfill-order', (event) => {
  const { orderId } = event.payload;

  // Update fulfillment state based on views of other nodes
  fulfillmentNode.state.set((state, views) => {
    // Collect data from multiple services in one operation
    const fulfillmentData = views.collect({
      orders: state => {
        const order = state[orderId];
        return order ? { order } : null;
      },
      inventory: state => ({
        products: state.products
      }),
      customers: state => {
        // We need the order first to get the customerId
        const order = views.orders.get(s => s[orderId]);
        if (!order) return null;
        return {
          customer: state.customers[order.customerId]
        };
      },
      shipping: state => ({
        rates: state.rates
      })
    });

    // Check if order exists
    if (!fulfillmentData.orders) {
      return state; // No change
    }

    const order = fulfillmentData.orders.order;
    const customer = fulfillmentData.customers?.customer;

    if (!customer) {
```



```
    return state; // No change, customer not found
  }

  // Check inventory availability
  const itemsAvailable = order.items.every(item => {
    const product =
fulfillmentData.inventory.products[item.productId];
    return product && product.stock >= item.quantity;
  });

  if (!itemsAvailable) {
    return state; // No change, items not available
  }

  // Get shipping option
  const shippingOption =
fulfillmentData.shipping.rates[customer.region].standard;

  // Return updated fulfillment state
  return {
    ...state,
    fulfillments: {
      ...state.fulfillments,
      [orderId]: {
        orderId,
        items: order.items,
        customer: {
          id: order.customerId,
          name: customer.name,
          address: customer.address
        },
        shippingMethod: shippingOption.method,
        shippingCost: shippingOption.cost,
        estimatedDelivery: calculateDelivery(customer.address,
shippingOption),
        status: "ready",
        createdAt: Date.now()
      },
    },
  };
});

// Check if fulfillment was created
const fulfillment = fulfillmentNode.state.get(state =>
state.fulfillments?.[orderId])
```

```
);  
  
if (!fulfillment) {  
  return {  
    success: false,  
    reason: "Could not create fulfillment"  
  };  
}  
  
// Continue to next step  
return initiateShipment;  
});
```

The Information System provides a complete model for managing, accessing, and transforming information through three complementary concepts:

- **State:** What nodes own and control
- **Context:** The causal flow of information through events
- **Views:** Windows into state owned by others

Each plays a distinct role, working together to create a powerful yet conceptually clean information model:

- State provides ownership and authority
- Context provides causality and correlation
- Views provide access and visibility

With enhanced view collection capabilities, the Information System enables complex, coordinated operations across node boundaries while maintaining clean separation of concerns and adhering to Happen's philosophy of radical simplicity.

# State Management

State management in Happen provides powerful capabilities through minimal abstractions. This guide explores patterns and approaches for effectively managing state across your Happen applications.

## Core Principles

Happen's approach to state management follows these foundational principles:

1. **Clear Ownership:** Each node owns and is responsible for its specific state
2. **Functional Transformations:** State changes occur through pure functional transformations
3. **Event-Driven Updates:** State typically changes in response to events
4. **Decentralized Intelligence:** Nodes make autonomous decisions based on their local state
5. **Composition Over Complexity:** Complex state management emerges from composing simple patterns

Under the hood, Happen uses NATS JetStream's Key-Value store to provide durable, distributed state storage with high consistency guarantees.

## Basic State Operations

### Accessing State

Happen provides a clean, functional approach to accessing state:

```
// Access the complete state
const orderState = orderNode.state.get();
```

```
// Access with transformation (focused state access)
const activeOrders = orderNode.state.get(state =>
  (state.orders || {}).filter(order => order.status === "active")
);
```

## Transforming State

State transformations use a functional approach:

```
// Transform state with a function
orderNode.state.set(state => {
  return {
    ...state,
    orders: {
      ...state.orders,
      "order-123": {
        ...state.orders["order-123"],
        status: "shipped",
        shippedAt: Date.now()
      }
    }
  };
});
```

## Composing Transformations

You can compose transformations for cleaner code:

```
// Define reusable transformers
const markShipped = (orderId) => (state) => ({
  ...state,
  orders: {
    ...state.orders,
    [orderId]: {
      ...state.orders[orderId],
      status: "shipped",
      shippedAt: Date.now()
    }
  }
});
```

```
});  
  
// Apply transformer  
orderNode.state.set(markShipped("order-123"));
```

## View System

Views provide a window into other nodes' state, enabling coordinated operations across node boundaries.

### Basic View Usage

```
// Transform state with views into other nodes' state  
orderNode.state.set((state, views) => {  
  // Get customer data through views  
  const customerData = views.customer.get(state =>  
    state.customers[order.customerId]  
  );  
  
  // Get inventory data through views  
  const inventoryData = views.inventory.get(state =>  
    state.products[order.productId]  
  );  
  
  // Return updated state that incorporates external data  
  return {  
    ...state,  
    orders: {  
      ...state.orders,  
      [order.id]: {  
        ...order,  
        canShip: inventoryData.inStock,  
        shippingAddress: customerData.address  
      }  
    }  
  };  
});
```

### Enhanced Collection

For more efficient collection of state from multiple nodes, use the collect pattern:

```
orderNode.state.set((state, views) => {
  // Collect state from multiple nodes in a single operation
  const data = views.collect({
    customer: state => ({
      name: state.customers[order.customerId].name,
      address: state.customers[order.customerId].address
    }),
    inventory: state => ({
      inStock: state.products[order.productId].quantity > 0,
      leadTime: state.products[order.productId].leadTime
    }),
    shipping: state => ({
      rates: state.ratesByRegion[customerRegion],
      methods: state.availableMethods
    })
  });

  // Use the collected data
  return {
    ...state,
    orders: {
      ...state.orders,
      [order.id]: {
        ...order,
        fulfillmentPlan: createFulfillmentPlan(data)
      }
    }
  };
});
```

This approach:

- Traverses the node graph only once
- Collects specific slices of state from each relevant node
- Transforms the state during collection
- Returns a unified object with all the collected data

# Scaling Patterns

As your application grows, these patterns help manage increasing state complexity.

## Selective State Access

Access only what you need to minimize memory footprint:

```
// Instead of loading the entire state
const allOrders = orderNode.state.get();

// Access only what you need
const orderCount = orderNode.state.get(state =>
  Object.keys(state.orders || {}).length
);
```

## Domain Partitioning

Partition state across multiple domain-specific nodes:

```
// Instead of one large node with all state
const monolithNode = createNode("monolith");

// Create domain-specific nodes
const orderNode = createNode("order-service");
const customerNode = createNode("customer-service");
const inventoryNode = createNode("inventory-service");
```

Each node maintains state for its specific domain, communicating via events when necessary.

## Event-Driven State Synchronization

Use events to communicate state changes between nodes:

```
// When state changes, broadcast an event
orderNode.on("update-order", event => {
  // Update local state
  orderNode.state.set(state => ({
    ...state,
    orders: {
      ...state.orders,
      [event.payload.orderId]: event.payload
    }
  }));

  // Broadcast change event
  orderNode.broadcast({
    type: "order-updated",
    payload: {
      orderId: event.payload.orderId,
      status: event.payload.status
    }
  });
});

// Other nodes maintain caches of relevant order data
dashboardNode.on("order-updated", event => {
  // Update local cache
  dashboardNode.state.set(state => ({
    ...state,
    orderCache: {
      ...state.orderCache,
      [event.payload.orderId]: {
        status: event.payload.status,
        updatedAt: Date.now()
      }
    }
  }));
});
```

## Chunked Processing

For large state objects, process in manageable chunks:



```
orderNode.on("process-large-dataset", async function* (event) {
  const { dataset } = event.payload;

  // Process in chunks
  for (let i = 0; i < dataset.length; i += 100) {
    const chunk = dataset.slice(i, i + 100);

    // Process this chunk
    for (const item of chunk) {
      processItem(item);
    }

    // Yield progress
    yield {
      progress: Math.round((i + chunk.length) / dataset.length *
100),
      processed: i + chunk.length
    };
  }

  return { completed: true };
});
```

## Advanced Patterns

### State Versioning

Track state versions for debugging and auditing:

```
// Modify state with version tracking
orderNode.state.set(state => {
  const newVersion = (state.version || 0) + 1;

  return {
    ...state,
    version: newVersion,
    orders: {
      ...state.orders,
      "order-123": {
        ...state.orders["order-123"]
      }
    }
  };
});
```

```

        status: "processing",
        updatedAt: Date.now(),
        version: newVersion
      }
    }
  };
});

```

## Computed State

Define computed values based on raw state:

```

orderNode.state.set((state, views) => {
  // Compute derived values
  const orderSummary = Object.values(state.orders ||
  {}).reduce((summary, order) => {
    summary.total += order.total;
    summary.count += 1;
    summary[order.status] = (summary[order.status] || 0) + 1;
    return summary;
  }, { total: 0, count: 0 });

  // Return state with computed values
  return {
    ...state,
    computed: {
      ...state.computed,
      orderSummary,
      averageOrderValue: orderSummary.count > 0 ?
        orderSummary.total / orderSummary.count : 0
    }
  };
});

```

## State-Based Event Generation

Generate events based on state conditions:

```

orderNode.state.set((state, views) => {

```

```
// Update order state
const updatedState = {
  ...state,
  orders: {
    ...state.orders,
    "order-123": {
      ...state.orders["order-123"],
      status: "payment-received"
    }
  }
};

// Check conditions
const order = updatedState.orders["order-123"];
const inventory = views.inventory.get(state =>
  state.products[order.productId]
);

// Emit events based on state conditions
if (order.status === "payment-received" && inventory.inStock) {
  orderNode.broadcast({
    type: "order-ready-for-fulfillment",
    payload: { orderId: "order-123" }
  });
}

return updatedState;
});
```

## NATS JetStream Key-Value Store Integration

Under the hood, Happen uses NATS JetStream's Key-Value store capability to provide durable, distributed state storage. The framework handles all the details of storing and retrieving state, ensuring:

1. **Persistence:** State is preserved even if nodes or processes restart
2. **Distributed Access:** State can be accessed across different processes and machines
3. **Consistency:** State updates maintain causal ordering

#### 4. **Performance:** High-performance storage with minimal overhead

This integration happens transparently - your code uses the same state API regardless of where nodes are running or how state is stored.

## Best Practices

### Keep State Focused

Each node should maintain state relevant to its domain:

```
// Too broad - mixing domains
const monolithNode = createNode("app");
monolithNode.state.set({
  orders: { /* ... */ },
  customers: { /* ... */ },
  inventory: { /* ... */ },
  shipping: { /* ... */ }
});

// Better - focused domains
const orderNode = createNode("orders");
orderNode.state.set({ orders: { /* ... */ } });

const customerNode = createNode("customers");
customerNode.state.set({ customers: { /* ... */ } });
```

### Use Immutable Patterns

Always update state immutably to maintain predictability:

```
// AVOID: Direct mutation
orderNode.state.set(state => {
  state.orders["order-123"].status = "shipped"; // Mutation!
  return state;
});
```

```
// BETTER: Immutable updates
orderNode.state.set(state => ({
  ...state,
  orders: {
    ...state.orders,
    "order-123": {
      ...state.orders["order-123"],
      status: "shipped"
    }
  }
}));
```

## Prefer Event Communication

Use events to communicate state changes between nodes:

```
// When state changes
orderNode.state.set(/* update state */);

// Broadcast the change
orderNode.broadcast({
  type: "order-status-changed",
  payload: {
    orderId: "order-123",
    status: "shipped"
  }
});
```

## Let Event Flow Drive State Changes

Instead of directly modifying state in response to external conditions, let events drive state changes:

```
// Respond to events with state transformations
orderNode.on("payment-received", event => {
  orderNode.state.set(state => ({
    ...state,
    orders: {
      ...state.orders,
```

```
[event.payload.orderId]: {  
  ...state.orders[event.payload.orderId],  
  status: "paid",  
  paymentId: event.payload.paymentId,  
  paidAt: Date.now()  
}  
}  
}));  
});
```

By focusing on clear ownership, functional transformations, and event-driven communication, you can create systems that are both powerful and maintainable. The integration with NATS JetStream's Key-Value store provides a robust foundation for distributed state management without adding complexity to your application code.

# Data Flow

Happen offers a unique approach to system design by providing two complementary layers that form a cohesive fabric for the movement, management, and manipulation of data: an event system and a state system. Rather than forcing you to choose between these approaches, Happen enables you to use both simultaneously, allowing you to leverage the strengths of each where appropriate.

## Two Complementary Layers

At the heart of Happen are two fundamentally different but complementary architectural layers:

### Event System: The Communication Layer

The event system forms the communication layer where nodes interact through message passing. Each event represents something that happened, and nodes react to these events by processing them and potentially emitting new events.

This layer excels at modeling:

- Distributed systems where components operate independently
- Real-time reactions to external stimuli
- Complex communication patterns with rich history tracking
- Systems where the sequence and timing of interactions matter

### State System: The Data Layer

The state system forms the data layer where each node manages its internal state. State evolves through well-defined transformations, and nodes can

access and modify their state in a consistent manner.

This layer excels at modeling:

- Entity lifecycles with clear states and transformations
- Consistent, atomic updates across multiple entities
- Business processes with well-defined rules
- Systems where the current state determines what can happen next

## Distinct APIs for Distinct Concerns

Happen provides separate APIs for each layer, making it clear which layer you're working with:

```
const orderNode = createNode("order-service");

// Event System: Process events through handlers
orderNode.on(type => type === "order-submitted", (event) => {
  validateOrder(event.payload);
  orderNode.broadcast({
    type: "order-validated",
    payload: {
      orderId: event.payload.orderId,
      items: event.payload.items
    }
  });
});

// State System: Transform state through the .state namespace
orderNode.state.set(state => {
  return {
    ...state,
    orders: {
      ...state.orders,
      "order-123": {
        ...state.orders["order-123"],
        status: "processing",
        processedAt: Date.now()
      }
    }
  }
});
```



```

    }
  };
});

```

This clear separation of concerns allows you to use the right tool for each task without artificial constraints.

## Layer Interaction

While the event and state layers are conceptually distinct, they interact seamlessly within nodes:

```

// Event processing triggering state changes
paymentNode.on(type => type === "payment-received", (event) => {
  const { orderId, amount } = event.payload;

  // Process the payment
  const result = processPayment(event.payload);

  // Update state in response to the event
  paymentNode.state.set(state => {
    const orders = state.orders || {};
    const order = orders[orderId] || {};

    return {
      ...state,
      orders: {
        ...orders,
        [orderId]: {
          ...order,
          paymentStatus: "paid",
          paymentId: result.transactionId,
          paidAt: Date.now()
        }
      }
    };
  });

  // Emit another event based on the state change
  paymentNode.broadcast({

```

```
    type: "payment-confirmed",
    payload: {
      orderId,
      transactionId: result.transactionId
    }
  });
});
```

This ability to seamlessly move between layers allows you to create systems where communication and state management work together naturally.

## State Operations

The state system provides powerful capabilities for managing state at different granularities.

## Individual Entity Operations

For focused state updates on specific entities:

```
// Individual entity transformation
orderNode.state.set(state => {
  const orders = state.orders || {};
  const order = orders["order-123"] || {};

  return {
    ...state,
    orders: {
      ...orders,
      "order-123": {
        ...order,
        status: "processing"
      }
    }
  };
});
```

## Multi-Entity Operations

## Multi-Entity Operations

For operations that need to maintain consistency across multiple entities:

```
// Multi-entity transformation
orderNode.state.set((state, views) => {
  // Get state from this node
  const orders = state.orders || {};
  const order = orders["order-123"];

  if (!order) return state;

  // Get state from inventory node through views
  const inventory = views.inventory.get();

  // Update both order and inventory
  const updatedInventory = updateInventory(inventory, order.items);

  // Return transformed state
  return {
    ...state,
    orders: {
      ...orders,
      "order-123": {
        ...order,
        status: "processing"
      }
    }
  };
});
```

## Functional Composition

The state system embraces functional composition as a core principle, allowing you to build complex transformations from simpler ones:

```
// Define transformation functions
const validateOrder = (state) => ({
  ...state,
  validated: true,
  validatedAt: Date.now()
});
```

```
});

const processPayment = (state) => ({
  ...state,
  paymentProcessed: true,
  paymentProcessedAt: Date.now()
});

const prepareShipment = (state) => ({
  ...state,
  shipmentPrepared: true,
  shipmentPreparedAt: Date.now()
});

// Apply composed transformation
orderNode.state.set(state => {
  // Get the order
  const orders = state.orders || {};
  const order = orders["order-123"] || {};

  // Base transformation
  const baseState = {
    ...order,
    status: "processing"
  };

  // Apply composed transformations
  const processedOrder = [validateOrder, processPayment,
    prepareShipment]
    .reduce((currentState, transform) => transform(currentState),
    baseState);

  // Return the updated state
  return {
    ...state,
    orders: {
      ...orders,
      "order-123": processedOrder
    }
  };
});
```

This functional approach allows for powerful, flexible composition without

requiring special helpers or syntax.

## When to Use Each Layer

While Happen allows you to freely use both layers, here are some guidelines for choosing the right approach:

Use the Event System when:

- You need to communicate between nodes
- Events arrive from external sources
- The history and sequence of interactions matter
- You need to broadcast information to multiple recipients
- You're modeling reactive behavior

Use the State System when:

- You need atomic updates to a node's state
- State consistency is critical
- You're working with well-defined entity lifecycles
- You need to validate state changes before applying them
- You're modeling procedural behavior

## The Best of Both Worlds

By providing both event and state layers, Happen allows you to build systems that are:

- Reactive to external events and stimuli
- Consistent in their state management
- Distributed across multiple components

- Predictable in their behavior
- Flexible in their evolution
- Comprehensible to developers

This dual-layer architecture gives Happen a unique power: the ability to create complex systems where communication and state management work together seamlessly, using the right approach for each concern.

# Temporal State

In Happen, the journey is just as important as the destination. Temporal State leverages NATS JetStream's powerful capabilities to provide access to your state across time, enabling powerful historical analysis and recovery with minimal complexity.

Temporal State gives nodes the ability to access and work with historical state—allowing you to see not just what your state *is* now, but what it *was* at important points in your application's history.

## How Temporal State Works with JetStream

At its core, Temporal State in Happen is built on NATS JetStream's key features:

1. **Durable Streams:** JetStream stores sequences of messages with configurable retention
2. **Key-Value Store:** Built on streams, preserves state changes as versioned entries
3. **Message Headers:** Carry metadata about events, including causal relationships
4. **Event Sourcing Pattern:** Natural event sourcing capabilities through message ordering

When an event flows through a node and modifies state, both the event and the resulting state change are recorded in JetStream:

```
// Conceptually, a temporal state snapshot includes:
{
  state: { /* state at this point in time */ },
  context: { // Essential event context
    id: 'evt-456', // The event that created this state
    causationId: 'evt-123', // What caused this event
  }
}
```

```

    correlationId: 'order-789', // Transaction this event belongs
  to
    sender: 'payment-service', // Node that sent the event
    timestamp: 1621452789000, // When it happened
    eventType: 'payment-processed' // Type of event
  }
}

```

This combination of event and state history provides a complete record of how your system evolved over time.

## JetStream Key-Value Store for Historical State

Happen uses JetStream's Key-Value store capabilities to implement Temporal State:

```

// Initialize with Temporal State configuration
const happen = initializeHappen({
  nats: {
    capabilities: {
      persistence: {
        enabled: true,
        keyValue: {
          enabled: true,
          buckets: {
            state: "happen-state",
            temporal: "happen-temporal"
          }
        }
      },
    },
    // Temporal state configuration
    temporal: {
      enabled: true,
      history: 100, // Keep 100 versions per key
      maxAge: "30d" // Keep history for 30 days
    }
  }
});

```



This configuration creates a specialized Key-Value bucket that preserves historical versions of state entries.

## Accessing Temporal State

Happen provides a clean and intuitive way to access historical state through the `.when()` function:

```
// Access state after a specific event
orderNode.state.when('evt-123', (snapshots) => {
  // Work with the complete historical snapshot
  const { state, context } = snapshots[0];

  console.log(`Order status was: ${state.orders["order-123"].status}`);
  console.log(`Event type: ${context.eventType}`);
  console.log(`Caused by: ${context.causationId}`);

  return processHistoricalState(state, context);
});
```

The `when` function follows Happen's pattern-matching approach, accepting either:

- A string event ID: 'evt-123'
- A function matcher: `eventId => eventId.startsWith('payment-')`

## Traversing Causal Chains

Since each snapshot includes context information, you can easily traverse causal chains using pattern matching:

```
// Find all states caused by a specific event
orderNode.state.when(
  event => event.causationId === 'evt-123',
  // ...
```

```
(snapshots) => {
  // Process all snapshots directly caused by evt-123
  const eventTypes = snapshots.map(snap =>
snap.context.eventType);

  console.log(`Event evt-123 caused these events:
${eventTypes.join(', ')}`);

  return analyzeEffects(snapshots);
}
);

// Or use correlation IDs to get entire transaction flows
orderNode.state.when(
  event => event.correlationId === 'order-789',
  (snapshots) => {
    // Process all snapshots in the transaction
    // Arrange by timestamp to see the sequence
    const eventSequence = [...snapshots]
      .sort((a, b) => a.context.timestamp - b.context.timestamp)
      .map(snap => snap.context.eventType);

    console.log(`Transaction order-789 flow: ${eventSequence.join('
→ ')}`);

    return createAuditTrail(snapshots);
  }
);
```

## Efficient Implementation through JetStream

Rather than storing complete copies of state for every event, Happen leverages JetStream's efficient storage capabilities:

- JetStream automatically compresses and deduplicates data
- Key revisions are tracked with minimal overhead
- The system intelligently manages resource usage
- Historical data is pruned based on configurable policies

This approach balances efficient storage with powerful historical access capabilities.

## Customizing Retention Policies

Happen allows you to customize which historical states are retained through JetStream's retention policies:

```
// Node with custom retention policy
const orderNode = createNode('order-service', {
  persistence: {
    temporal: {
      history: 50,           // Keep up to 50 versions
      maxAge: "7d",         // Keep history for 7 days
      subject: "order.*"    // Only track states for order events
    }
  }
});
```

This gives you control over:

- How many historical versions to keep
- How long to keep historical versions
- Which events should create temporal snapshots

## Event Sourcing

Temporal State makes implementing the Event Sourcing pattern remarkably straightforward:

```
// Rebuild state at a specific point in time
orderNode.state.when('evt-123', (snapshot) => {
  // Replace current state with historical state
  orderNode.state.set(() => snapshot.state);
});
```

```
// Let the system know we rebuilt state
orderNode.broadcast({
  type: 'state-rebuilt',
  payload: {
    fromEvent: snapshot.context.id,
    timestamp: snapshot.context.timestamp
  }
});

return { rebuilt: true };
});
```

## Recovery and Resilience

Temporal State provides natural resilience capabilities:

```
// After node restart, recover from latest known state
function recoverLatestState() {
  // Find the latest event we processed before crashing
  const latestEventId = loadLatestEventIdFromDisk();

  if (latestEventId) {
    // Recover state from that point
    orderNode.state.when(latestEventId, (snapshot) => {
      if (snapshot) {
        // Restore state
        orderNode.state.set(() => snapshot.state);
        console.log('State recovered successfully');
      }
    });
  }
}
```

## Benefits of JetStream-Powered Temporal State

Happen's JetStream-based approach to Temporal State offers several key advantages:

1. **Durable History:** State history persists even across system restarts
2. **Efficient Storage:** JetStream optimizes storage with minimal overhead
3. **Causal Tracking:** Event relationships are preserved throughout history
4. **Tunable Retention:** Customize retention based on your specific needs
5. **Cross-Node Consistency:** Historical state is consistent across node instances
6. **Performance:** JetStream provides high-performance access to historical data
7. **Scalability:** Works from single-process to globally distributed systems

By leveraging NATS JetStream's capabilities, Happen provides powerful temporal state features without the complexity typically associated with time-travel debugging and historical analysis.

Adding the dimension of time to your application's data model, Temporal State opens up powerful capabilities with minimal added complexity. Since it builds on NATS JetStream's existing features, it provides powerful capabilities that feel natural and integrated.

With Temporal State, your applications gain new powers for auditing, debugging, analysis, recovery, and understanding—all while maintaining Happen's commitment to simplicity and power through minimal primitives.

# The Unified Event Space

In Happen, events flow seamlessly across process, network, and environment boundaries through a unified NATS backbone. This approach eliminates the artificial boundaries typically found in distributed systems, enabling you to build applications that span multiple processes and environments with the same simplicity as local ones.

## Beyond Boundaries

Happen's unified event space means you write code the same way regardless of where nodes are deployed:

```
// The exact same code works whether nodes are:  
// - In the same process  
// - In different processes on the same machine  
// - Distributed across different machines  
// - In completely different environments (server, browser, edge)  
orderNode.send(inventoryNode, {  
  type: "check-inventory",  
  payload: { orderId: "123", items: [...] }  
});
```

This isn't just an API convenience—it's a fundamental design principle that creates location transparency throughout your system. Nodes can be deployed wherever makes the most sense for your architecture, and your code remains unchanged.

## How the Unified Event Space Works

At the heart of Happen's unified event space is the NATS messaging system with its JetStream persistence layer. When an event crosses boundaries:

1. The event is automatically serialized using MessagePack

2. NATS delivers it to the appropriate destination
3. The event is deserialized on arrival
4. The receiving node processes it using standard event handlers

All of this happens transparently—your code simply uses `send()` and `on()` without needing to know whether communication is local or remote.

## Cross-Boundary Serialization

When events cross process or network boundaries in Happen, they are efficiently handled through a streamlined serialization strategy:

### JSON Interface, Binary Transport

Happen provides a natural JSON-like interface for developers while using efficient binary serialization under the hood:

- **Developer Experience:** Work with standard JavaScript objects in your code
- **Transport Efficiency:** MessagePack binary format used for actual transmission
- **Transparent Conversion:** Serialization/deserialization happens automatically
- **No Schema Required:** Maintains Happen's schema-free flexibility

This approach gives you the best of both worlds - the simplicity of working with native JavaScript objects and the efficiency of binary transport.

## Preserving Causality Across Boundaries

One of the most powerful aspects of Happen's unified event space is how it maintains causality across boundaries. Every event carries its complete

causal context:

```
// When an event crosses a boundary, its context is preserved
{
  type: "inventory-reserved",
  payload: {
    orderId: "order-123",
    items: [/* ... */]
  },
  context: {
    causal: {
      id: "evt-789", // Unique event ID
      sender: "inventory-node", // Originating node
      causationId: "evt-456", // Event that caused this one
      correlationId: "txn-123", // Transaction this belongs to
      path: [ // Complete journey
        ["user-node", "order-node"],
        ["order-node", "inventory-node"]
      ]
    }
  }
}
```

This causal context creates a complete web of relationships that spans your entire distributed system.

## Unified Configuration

Happen provides a clean, explicit approach to configuration that separates environment concerns from application logic:

```
// Initialize Happen with NATS as the universal backbone
const happen = initializeHappen({
  // NATS configuration
  nats: {
    // Connection configuration
    connection: {
      // Server environment (direct NATS)
      server: {
```



```

        servers: ['nats://server:4222'],
        jetstream: true
    },
    // Browser environment (WebSocket)
    browser: {
        servers: ['wss://server:8443'],
        jetstream: true
    }
},

// Enable key features
capabilities: {
    // Persistence through JetStream
    persistence: {
        enabled: true,
        // Use Key-Value store for state
        keyValue: {
            enabled: true,
            buckets: {
                state: "happen-state",
                temporal: "happen-temporal"
            }
        }
    },

    // Delivery guarantees
    delivery: {
        exactlyOnce: true,
        deduplication: true
    }
}
});

```

## Node-Level Configuration

Individual nodes can specify their own delivery requirements:

```

// Node with specific reliability requirements
const paymentNode = createNode("payment-service", {
    // Delivery guarantees for this specific node
    deliver: {

```

```
        exactlyOnce: true,  
        acknowledge: true,  
        timeout: 5000, // ms  
        retries: 3  
    }  
});
```

## Large Payload Management

For handling large data across boundaries, Happen leverages NATS Key-Value store to provide a global namespace:

### The Global Namespace

```
// Store large data  
const key = `data:${generateId()}`;  
node.global.set(key, largeData);  
  
// Reference in events  
node.send(targetNode, {  
  type: "process-data",  
  payload: { dataKey: key }  
});  
  
// Retrieve on the other side  
const data = await targetNode.global.get(event.payload.dataKey);
```

## Handling Network Realities

Real-world networks are unreliable. Happen's NATS-based transport layer addresses these realities directly:

### Persistent Delivery

Events can be stored in JetStream's durable storage until successfully

processed, ensuring:

- **Durability:** Events survive process restarts
- **Guaranteed Delivery:** Events reach their destination even after network issues
- **Natural Backpressure:** JetStream provides backpressure for overloaded consumers

## Network Partitions

During network partitions:

- Nodes continue operating within their connected segments
- Events destined for disconnected nodes are stored in JetStream
- When connectivity resumes, stored events are delivered
- Causality is preserved throughout this process

This happens automatically through JetStream's durable storage capabilities.

## Cross-Environment Communication

The unified transport system makes cross-environment communication seamless:

```
// Server-side node sending to browser client
serverNode.on("update-client", (event) => {
  // Use the standard send method - routing happens automatically
  serverNode.send(browserNode, {
    type: "data-updated",
    payload: event.payload.data
  });
});
```

# Benefits of the Unified Event Space

This approach to distributed events creates powerful capabilities:

1. **Location Transparency:** Your code doesn't need to know or care where nodes are deployed
2. **Protocol Flexibility:** The right protocol is selected automatically for each environment
3. **Environment Adaptability:** Seamlessly bridge between server, browser, and edge environments
4. **Deployment Freedom:** Components can move between environments by simply changing deployment configuration
5. **Resilient Architecture:** Systems naturally handle network fluctuations through JetStream
6. **Future Extensibility:** Easily adopt new NATS capabilities as they become available

By providing a truly unified event space powered by NATS, Happen enables you to focus on your application's domain rather than the complexities of distributed systems.

# Identity & Auth

Happen provides the minimal, essential tools that enable developers to build their own identity models. Here's what that looks like:

## 1. Node Identity

Every node in Happen possesses a unique identity established at creation:

```
const orderNode = createNode('order-service');
```

This creates a node with a cryptographic identity that:

- Is established automatically at node creation
- Is used to verify the origin of events
- Remains consistent throughout the node's lifecycle
- Can be referenced by other nodes

Node identity is foundational to Happen's event flow, enabling verifiable communication between components without requiring elaborate identity infrastructure.

## 2. Origin Context

Events in Happen carry origin information in their context, providing a clear record of where events came from:

```
{  
  type: "order-created",  
  payload: {  
    // Domain-specific data  
    orderId: "order-123",  
    ...  
  }  
}
```

```

    items: [/* items data */]
  },
  context: {
    causal: {
      id: "evt-789",
      sender: "checkout-node",
      causationId: "evt-456",
      correlationId: "txn-123"
    },
    origin: {
      nodeId: "checkout-node",      // The immediate sender node
      (added automatically)
      sourceId: "user-456",        // Original source (optional,
      application-provided)
      sourceType: "user"           // Type of source (optional)
    }
  }
}

```

The origin context provides:

- **nodeId**: The node that immediately sent the event (added automatically)
- **sourceId**: The original source identity (e.g., user ID, service ID)
- **sourceType**: The type of source (e.g., "user", "system", "service")

This minimal but powerful primitive allows events to carry essential identity information without prescribing how that information should be structured or used.

### 3. Acceptance Controls

Nodes can specify which origins they're willing to accept events from:

```

const paymentNode = createNode("payment-service", {
  acceptFrom: [
    "checkout-node",      // Accept from specific node
    "user:admin",         // Accept from specific user type/role
    "order-*"             // Accept from nodes matching pattern
  ]
})

```

```
    }  
  });
```

This declarative approach provides a simple way to control event flow based on origin, without requiring complex configuration or policy engines.

## How Happen Enables Identity Models

By providing these blocks rather than a complete identity system, Happen enables developers to:

### Build Custom Identity Models

```
// Example: Multi-tenant identity model  
function createTenantNode(tenantId, nodeId) {  
  return createNode(nodeId, {  
    // Only accept events from the same tenant  
    acceptFrom: [`tenant:${tenantId}:*`]  
  });  
}  
  
// When sending events, include tenant context  
sourceNode.send(targetNode, {  
  type: "process-data",  
  payload: { /* data */ },  
  context: {  
    origin: {  
      sourceId: `tenant:${tenantId}:user-123`,  
      sourceType: "user"  
    }  
  }  
});
```

### Create Domain-Specific Authorization

```
// Custom authorization using the Event Continuum  
orderNode.on("update-order", (event, context) => {  
  // Extract origin information
```

```
// Extract origin information
const { sourceId, sourceType } = event.context.origin;

// Get the order
const order = getOrder(event.payload.orderId);

// Check if the source has permission to update this order
if (sourceType === "user" && order.customerId !== sourceId) {
  return {
    success: false,
    reason: "unauthorized",
    message: "Users can only update their own orders"
  };
}

// Continue with authorized flow
return processOrderUpdate;
});
```

## Implement Delegation Patterns

```
// Forward events while preserving original source
apiNode.on("create-order", (event, context) => {
  // Preserve the original source when forwarding
  return apiNode.send(orderNode, {
    type: "process-order",
    payload: event.payload,
    context: {
      origin: {
        // Keep original source information
        sourceId: event.context.origin.sourceId,
        sourceType: event.context.origin.sourceType
      }
    }
  });
});
```

## Embracing Causality

The identity system in Happen is designed to work seamlessly with its causal



model:

- Each event naturally references its sender node
- The causal chain provides a complete history of who did what
- Origin information flows through event chains
- The combination creates a natural audit trail

This integration means that identity becomes an inherent property of the event flow rather than a separate concern requiring special handling.

## The Power of Minimalism

By providing just the essential identity apparatus, Happen enables:

1. **Simple Identity Tracking:** Basic origin tracking with zero configuration
2. **Powerful Custom Models:** Build sophisticated identity systems when needed
3. **Domain-Specific Authorization:** Implement authorization that matches your domain
4. **Clean Mental Model:** Identity as a natural property of events

# Security

Happen provides the minimal, essential tools that enable developers to build their own security models.

## 1. Event Identity and Integrity Tooling

**Event ID Generation:** Every event in Happen automatically receives a unique, cryptographically secure identifier generated by the framework.

**Automatic Event Hash:** The framework automatically calculates and includes a cryptographic hash of each event's contents in its causal context:

```
// Example of what an event looks like internally
{
  type: "payment-processed",
  payload: { amount: 100 },
  context: {
    causal: {
      id: "evt-789",           // Unique ID
      hash: "a1b2c3...",     // Cryptographic hash
      sender: "payment-node",
      causationId: "evt-456",
      correlationId: "order-123"
    }
  }
}
```

This hash is calculated from the canonicalized event (without the hash itself) using a hashing algorithm provided by the runtime environment.

## 2. Node Identity

**Identity Container:** Every node in Happen has an inherent identity established at creation:

```
// Node identity is established automatically
const paymentNode = createNode('payment-service');
```

This identity is used to track the origin of events and forms the foundation of security verification.

**Identity Propagation:** Node identity is automatically included in the causal context of emitted events:

```
// The framework automatically includes the sender information
// in the causal context of every event
orderNode.send(paymentNode, {
  type: "process-payment",
  payload: { amount: 100 }
});

// Resulting event includes sender identity
// {
//   type: "process-payment",
//   payload: { amount: 100 },
//   context: {
//     causal: {
//       sender: "order-service", // Added automatically
//       ...
//     }
//   }
// }
```

**Origin Context:** For tracking the original source of events (not just the immediate sender):

```
// Adding origin information to events
orderNode.send(paymentNode, {
  type: "process-payment",
  payload: { amount: 100 },
  context: {
    origin: {
      sourceId: "user-123", // Original source (e.g., user ID)
      sourceType: "user" // Type of source
    }
  }
});
```

```
    {  
    }  
  });
```

This allows developers to build their own identity models on top of Happen.

### 3. Causal Chain

The causal relationships between events provide a natural foundation for security verification:

```
// Events naturally form a causal chain  
event.context.causal = {  
  id: "evt-789",  
  sender: "user-service",  
  causationId: "evt-456",    // Links to the event that caused this  
  one  
  correlationId: "session-123" // Groups related events  
};
```

This causality enables:

- Tracking the complete flow of events
- Verifying that events occurred in the expected sequence
- Building audit trails based on causal relationships
- Detecting replay or injection attacks

### 4. The Event Continuum for Verification

The Event Continuum provides a natural place to implement security verification:

```
// Security verification using the Event Continuum  
paymentNode.on("process-payment", function verifyAndProcess(event,  
  context) {
```

```
CONTEXT) {
  // Extract security information
  const { signature } = event.context.integrity || {};
  const { hash } = event.context.causal;

  // Verify using runtime crypto
  if (signature) {
    const senderPublicKey =
      getPublicKey(event.context.causal.sender);
    if (!verifySignature(hash, signature, senderPublicKey)) {
      return {
        success: false,
        reason: "invalid-signature"
      };
    }
  }

  // Continue to payment processing
  return processPayment;
});
```

This allows security verification to integrate seamlessly with the normal event flow.

## How These Enable Security

By providing building blocks rather than implementations, Happen enables developers to build their own security models:

### Integrity Verification

```
// Using the automatic hash for integrity verification
paymentNode.on("process-payment", (event, context) => {
  // Hash verification happens automatically by the framework

  // Can perform additional custom verification if needed
  if (!customVerification(event)) {
    return { success: false, reason: "verification-failed" };
  }
});
```

```
// Continue processing
return processPayment;
});
```

## Digital Signatures and Tamper Evidence

```
// Signing events using runtime crypto
orderNode.on("create-order", (event, context) => {
  // Generate order and prepare payment event
  const paymentEvent = {
    type: "process-payment",
    payload: {
      orderId: generateOrderId(),
      amount: calculateTotal(event.payload.items)
    },
    context: {
      // Add signature using runtime crypto
      integrity: {
        signature: signWithRuntimeCrypto(
          event.context.causal.hash,
          getPrivateKey()
        )
      }
    }
  };

  // Send signed event
  return orderNode.send(paymentNode, paymentEvent);
});
```

## Authorization Patterns

```
// Authorization using the Event Continuum
resourceNode.on("access-resource", (event, context) => {
  // Extract source information
  const { sourceId, sourceType } = context.origin || {};

  // Check permissions
  if (!hasPermission(sourceId, event.payload.resourceId, "read")) {
    return {
```

```

        success: false,
        reason: "unauthorized",
        message: "User does not have permission to access this
resource"
    };
}

// Continue to resource access
return provideResource;
});

```

## Multi-tenant Isolation

```

// Tenant isolation using origin information
dataNode.on("query-data", (event, context) => {
    // Extract tenant from origin
    const tenantId =
extractTenantFromSource(context.origin?.sourceId);

    // Enforce tenant isolation
    const tenantScopedQuery = {
        ...event.payload.query,
        tenantId // Add tenant filter to query
    };

    // Execute scoped query
    const results = executeQuery(tenantScopedQuery);

    return { results };
});

```

## End-to-End Encryption

```

// Encrypting sensitive payload data
userNode.on("send-sensitive-message", (event, context) => {
    // Get recipient's public key
    const recipientPublicKey =
getPublicKey(event.payload.recipientId);

    // Create encrypted message
    const encryptedMessage = encryptWithRuntimeCrypto(

```

```
const encryptedMessage = encryptWithPublicKey(\n  event.payload.sensitiveContent,\n  recipientPublicKey\n);\n\n// Send message with encrypted content\nreturn userNode.send(messagingNode, {\n  type: "deliver-message",\n  payload: {\n    recipientId: event.payload.recipientId,\n    encryptedContent: encryptedMessage\n  }\n});\n});
```

## Minimal but Powerful

The key is to expose the right features in the core event system, without baking in specific security implementations by giving developers all the building blocks they need without restricting their options.

These features enable developers to implement:

- Digital signatures for event verification
- Role-based access control for node interactions
- Attribute-based access policies
- Multi-layer security models
- Custom authentication mechanisms
- Auditing and compliance systems
- End-to-end encryption for sensitive data

While keeping the core framework minimal and focused on its primary responsibility: the flow of events between nodes.



# Error Handling

Error handling implements robust recovery patterns without introducing new primitives beyond Nodes and Events.

## Core Principles

Happen's approach to error handling is built on these fundamental principles:

1. **Errors as Flow Branches:** Errors are treated as natural branches in the event flow
2. **Functional Error Handling:** Error handlers are just functions that can be returned
3. **Context for Error Information:** Error details flow through context between functions
4. **Causal Error Tracking:** Errors maintain causal relationships like normal events
5. **Decentralized Recovery:** Nodes make local recovery decisions when possible

This approach achieves remarkable power with minimal complexity by leveraging existing primitives rather than introducing special error-handling constructs.

## The Functional Error Model

In Happen, error handling is fully integrated into the Event Continuum through function returns:

```
// Register an event handler
orderNode.on("process-order", function validateOrder(event,
context) {
    // Validate the order
```

```
// validate the order
const validation = validateOrderData(event.payload);

if (!validation.valid) {
  // Return error handler function on validation failure
  return handleInvalidOrder;
}

// Proceed with valid order
context.validated = true;
return processOrder;
});

// Error handler function
function handleInvalidOrder(event, context) {
  // Log the validation failure
  logValidationFailure(event.payload);

  // Return error result
  return {
    success: false,
    reason: "validation-failed",
    details: "Invalid order data"
  };
}
```

This functional approach means:

1. Errors are just another branch in the flow
2. Error handlers are regular functions
3. Error information flows through context
4. No special syntax or constructs are needed

## Error Propagation and Context

Error information naturally flows through context:

```
function processPayment(event, context) {
  try {
```

```
// Process the payment
const paymentResult = chargeCustomer(event.payload.payment);

if (!paymentResult.success) {
  // Store error details in context
  context.error = {
    code: paymentResult.code,
    message: paymentResult.message,
    timestamp: Date.now(),
    correlationId: event.context.causal.correlationId
  };

  return handlePaymentFailure;
}

// Success path
return createShipment;
} catch (error) {
  // Unexpected error - store and return handler
  context.error = {
    unexpected: true,
    message: error.message,
    stack: error.stack,
    timestamp: Date.now()
  };

  return handleUnexpectedError;
}

function handlePaymentFailure(event, context) {
  // Access error from context
  const { error } = context;

  // Log structured error
  logError("Payment failed", error);

  // Return result with error details
  return {
    success: false,
    reason: "payment-failed",
    details: error.message,
    code: error.code
  };
}
```

This approach allows error information to flow naturally through the event chain, maintaining complete context for diagnostics and recovery.

## Recovery Patterns

Happen's functional approach enables powerful recovery patterns through composition.

### Retry Pattern

```
function processPayment(event, context) {
  // Initialize retry count
  context.retryCount = context.retryCount || 0;

  try {
    // Attempt payment processing
    const result = chargeCustomer(event.payload.payment);

    // Success - continue to shipping
    context.payment = result;
    return createShipment;
  } catch (error) {
    // Store error
    context.lastError = error;

    // Increment retry count
    context.retryCount++;

    // Determine if we should retry
    if (context.retryCount < 3) {
      console.log(`Retrying payment (${context.retryCount}/3)...`);

      // Return self to retry
      return processPayment;
    }

    // Too many retries
    return handlePaymentFailure;
  }
}
```

```
}
```

## Circuit Breaker Pattern

```
// Shared circuit state (could be in external store)
const circuits = {
  payment: {
    failures: 0,
    status: "closed", // closed, open, half-open
    lastFailure: null
  }
};

function processPayment(event, context) {
  const circuit = circuits.payment;

  // Check if circuit is open
  if (circuit.status === "open") {
    return {
      success: false,
      reason: "service-unavailable",
      message: "Payment service is temporarily unavailable"
    };
  }

  try {
    // Process payment
    const result = chargeCustomer(event.payload.payment);

    // Success - reset circuit if in half-open state
    if (circuit.status === "half-open") {
      circuit.status = "closed";
      circuit.failures = 0;
    }

    // Continue with success
    context.payment = result;
    return createShipment;
  } catch (error) {
    // Update circuit state
    circuit.failures++;
    circuit.lastFailure = Date.now();
  }
}
```

```
// Check threshold for opening circuit
if (circuit.status === "closed" && circuit.failures >= 5) {
  // Open the circuit
  circuit.status = "open";

  // Schedule reset to half-open
  setTimeout(() => {
    circuit.status = "half-open";
  }, 30000); // 30 second timeout
}

// Return error result
return {
  success: false,
  reason: "payment-failed",
  message: error.message,
  circuitStatus: circuit.status
};
}
```

## Fallback Pattern

```
function processPayment(event, context) {
  try {
    // Try primary payment processor
    const result =
primaryPaymentProcessor.charge(event.payload.payment);
    context.payment = result;
    return createShipment;
  } catch (error) {
    // Store error for context
    context.primaryError = error;

    // Return fallback function
    return useBackupPaymentProcessor;
  }
}

function useBackupPaymentProcessor(event, context) {
  console.log("Primary payment processor failed. Using backup.");
```

```
try {
  // Try fallback processor
  const result =
    backupPaymentProcessor.charge(event.payload.payment);

  // Success - continue normal flow
  context.payment = result;
  context.usedFallback = true;
  return createShipment;
} catch (error) {
  // Both processors failed
  return {
    success: false,
    reason: "payment-failed",
    message: "Both primary and backup payment processors failed",
    primaryError: context.primaryError.message,
    backupError: error.message
  };
}
}
```

## Error Events as First-Class Citizens

In Happen, errors can also be treated as normal events, enabling system-wide error handling:

```
// When detecting an error, broadcast an error event
paymentNode.on("process-payment", (event, context) => {
  try {
    // Process payment
    const result = processPayment(event.payload);
    return { success: true, transactionId: result.id };
  } catch (error) {
    // Broadcast error event
    paymentNode.broadcast({
      type: "payment.error",
      payload: {
        orderId: event.payload.orderId,
        error: {
          message: error.message,
          code: error.code || "UNKNOWN"
        }
      }
    });
  }
});
```

```

        },
        timestamp: Date.now()
      }
    });

    // Return failure to the original sender
    return {
      success: false,
      reason: "payment-error",
      message: error.message
    };
  }
});

// Error monitoring node can observe all error events
monitoringNode.on(type => type.endsWith(".error"), (event) => {
  // Extract service name from event type
  const service = event.type.split('.')[0];

  // Record error
  recordServiceError(service, event.payload);

  // Check thresholds for alerts
  checkErrorThresholds(service);

  return { monitored: true };
});

```

## Distributed Error Handling

When errors cross node boundaries, they naturally maintain their causal context:

```

// Node A: Original error occurs
nodeA.on("process-task", async (event) => {
  try {
    // Process task
    const result = await processTask(event.payload);
    return { success: true, result };
  } catch (error) {
    // Return error result
    return {

```



```

        success: false,
        reason: "task-failed",
        error: error.message,
        timestamp: Date.now()
    };
    }
    });

// Node B: Handles failure from Node A
nodeB.on("orchestrate-workflow", async (event) => {
    // Call Node A
    const result = await nodeB.send(nodeA, {
        type: "process-task",
        payload: event.payload.taskData
    }).return();

    // Check for error
    if (!result.success) {
        // The error from Node A is now available in Node B
        // with the causal chain intact

        // Handle error based on reason
        if (result.reason === "task-failed") {
            return handleTaskFailure;
        }

        return {
            success: false,
            reason: "workflow-failed",
            cause: result.reason,
            error: result.error
        };
    }

    // Continue with success path
    return continueWorkflow;
});

```

## Supervisor Pattern

For system-wide resilience, you can create supervisor nodes that monitor and

## manage error recovery:

```
// Create a supervisor node
const supervisorNode = createNode("system-supervisor");

// Monitor error events across the system
supervisorNode.on(type => type.endsWith(".error"), (event) => {
  // Extract service from event type
  const service = event.type.split('.')[0];

  // Track error frequency
  supervisorNode.state.set(state => {
    const services = state.services || {};
    const serviceState = services[service] || { errors: 0,
lastError: 0 };

    return {
      ...state,
      services: {
        ...services,
        [service]: {
          ...serviceState,
          errors: serviceState.errors + 1,
          lastError: Date.now()
        }
      }
    };
  });

  // Check for restart threshold
  const serviceState = supervisorNode.state.get(state =>
    (state.services && state.services[service]) || { errors: 0 }
  );

  if (serviceState.errors >= 5) {
    // Too many errors - trigger restart
    supervisorNode.broadcast({
      type: "service.restart",
      payload: {
        service,
        reason: "excessive-errors",
        count: serviceState.errors
      }
    });
  }
});
```

```

    // Reset error count
    supervisorNode.state.set(state => ({
      ...state,
      services: {
        ...state.services,
        [service]: {
          ...state.services[service],
          errors: 0,
          lastRestart: Date.now()
        }
      }
    }));
  }

  return { monitored: true };
});

```

## Composing Error Handling with Normal Flow

Error handling in Happen integrates seamlessly with normal event processing:

```

// Complete order processing flow with error handling
orderNode.on("process-order", validateOrder);

function validateOrder(event, context) {
  // Validate order
  const validation = validateOrderData(event.payload);

  if (!validation.valid) {
    return handleInvalidOrder;
  }

  // Store validation result
  context.validatedOrder = {
    ...event.payload,
    validated: true,
    validatedAt: Date.now()
  };

  // Continue to inventory check
}

```

```
    return checkInventory;
  }

  function checkInventory(event, context) {
    // Check inventory
    const inventoryResult =
    checkInventoryLevels(context.validatedOrder.items);

    if (!inventoryResult.available) {
      // Store inventory problem
      context.inventoryIssue = inventoryResult;
      return handleInventoryShortage;
    }

    // Continue to payment
    return processPayment;
  }

  function processPayment(event, context) {
    try {
      // Process payment
      const paymentResult = chargeCustomer(
        context.validatedOrder.payment,
        calculateTotal(context.validatedOrder.items)
      );

      if (!paymentResult.success) {
        context.paymentIssue = paymentResult;
        return handlePaymentFailure;
      }

      // Store payment result
      context.payment = paymentResult;

      // Continue to shipment creation
      return createShipment;
    } catch (error) {
      context.error = error;
      return handleUnexpectedError;
    }
  }

  function createShipment(event, context) {
    // Create shipment
    const shipment = createShipmentRecord({
      context.validatedOrder.items
```

```
    order: context.validatedOrder,
    payment: context.payment
  });

  // Return success result
  return {
    success: true,
    orderId: context.validatedOrder.id,
    shipmentId: shipment.id,
    trackingNumber: shipment.trackingNumber
  };
}

// Error handlers
function handleInvalidOrder(event, context) {
  return {
    success: false,
    reason: "validation-failed",
    details: "Invalid order data"
  };
}

function handleInventoryShortage(event, context) {
  const { inventoryIssue } = context;

  // Try to get availability estimate
  const availability =
    estimateAvailability(inventoryIssue.missingItems);

  return {
    success: false,
    reason: "inventory-shortage",
    unavailableItems: inventoryIssue.missingItems,
    estimatedAvailability: availability
  };
}

function handlePaymentFailure(event, context) {
  return {
    success: false,
    reason: "payment-failed",
    details: context.paymentIssue.message,
    code: context.paymentIssue.code
  };
}
```

```
function handleUnexpectedError(event, context) {  
  // Log unexpected error  
  logError("Unexpected error processing order", context.error);  
  
  return {  
    success: false,  
    reason: "system-error",  
    message: "An unexpected error occurred"  
  };  
}
```

Treating errors as branches in the functional flow provides sophisticated error handling capabilities without introducing special constructs.

Key takeaways:

1. Use Function Returns for Error Flow: Return error handler functions to handle errors
2. Leverage Context: Store error information in the context object for diagnostics
3. Implement Recovery Patterns: Build error recovery using function composition
4. Apply Resilience Patterns: Implement retry, circuit breakers, and fallbacks as needed
5. Decentralized Recovery: Let nodes make local decisions about error handling

# Network Resilience

In distributed systems, network failures are inevitable. Nodes become unreachable, connections fail, and partitions occur. Happen provides comprehensive resilience capabilities through NATS and JetStream, creating a system that maintains causal integrity even during the most challenging network disruptions.

Happen's approach to network resilience stays true to its philosophy of radical simplicity by leveraging NATS' battle-tested features:

- **Durable Storage:** Events are preserved in JetStream before delivery attempts
- **Guaranteed Delivery:** At-least-once and exactly-once delivery semantics
- **Automatic Reconnection:** NATS clients intelligently reconnect after network failures
- **Message Replay:** Unacknowledged messages are automatically replayed when connections recover
- **Cross-Region Replication:** Messages remain available across geographic regions
- **Causal Ordering:** Events maintain their causal relationships throughout disruptions

This creates a self-healing system that requires minimal configuration while providing enterprise-grade resilience.

## Comprehensive Resilience Features

Happen combines multiple NATS capabilities to solve the full spectrum of network resilience challenges:

### 1. Persistent Message Storage

Events are automatically stored in NATS JetStream streams before delivery is attempted:

```
// When a node sends an event (internal implementation using NATS)
function sendEvent(targetNode, event) {
  // FIRST: Publish to JetStream stream with the event's ID as
  message ID
  // This happens automatically

  // THEN: Deliver to recipient
  // If successful, the recipient acknowledges
  // If unsuccessful, the event remains in JetStream for later
  delivery
}
```

This ensures events survive process crashes, network failures, and other disruptions.

## 2. Delivery Guarantees

Happen supports both at-least-once and exactly-once delivery semantics:

```
// Configure delivery guarantees at system level
const happen = initializeHappen({
  nats: {
    capabilities: {
      delivery: {
        // Choose your delivery semantics
        mode: "exactly-once", // or "at-least-once"
        deduplication: true,
        deduplicationWindow: "5m" // 5 minute window
      }
    }
  }
});

// Or configure per node
const paymentNode = createNode("payment-service", {
  delivery: {
    mode: "exactly-once",
```



```
    acknowledge: true
  }
});
```

With exactly-once delivery, even if messages are redelivered due to network issues, they'll only be processed once.

### 3. Automatic Reconnection

NATS clients automatically attempt to reconnect when network connections fail:

```
// Configure reconnection behavior
const happen = initializeHappen({
  nats: {
    connection: {
      // Reconnection settings
      reconnect: true,
      reconnectTimeWait: 2000, // 2 seconds between attempts
      maxReconnectAttempts: -1 // Unlimited reconnect attempts
    }
  }
});
```

During reconnection attempts, outbound messages are queued locally until the connection is restored.

### 4. Flow Control and Backpressure

NATS provides natural backpressure mechanisms that prevent overwhelming recipients during recovery:

```
// Configure flow control
const happen = initializeHappen({
  nats: {
    capabilities: {
      flowControl: {
```

```
        enabled: true,  
        maxPending: 256 * 1024 // 256KB of pending messages  
    }  
}  
}  
});
```

This ensures that when a node recovers after network issues, it won't be flooded with a sudden burst of messages.

## 5. Multi-Region Resilience

Happen leverages NATS SuperClusters to provide cross-region resilience:

```
// Configure multi-region operation  
const happen = initializeHappen({  
  nats: {  
    // SuperCluster configuration  
    connection: {  
      servers: ['nats://local-region:4222'],  
      jetstream: true  
    },  
    // Region configuration  
    regions: {  
      primary: "us-west",  
      replicas: ["us-east", "eu-central"],  
      // Configuration for handling cross-region communication  
      coordination: {  
        strategy: "primary-replica",  
        conflictResolution: "last-writer-wins"  
      }  
    }  
  }  
});
```

This enables:

- **Geographic Redundancy:** System continues operating even if an entire region fails

- **Data Locality:** Process data in the optimal region for performance and compliance
- **Disaster Recovery:** Automatic failover and recovery between regions

## 6. Causal Event Recovery

When network connections recover, events are replayed while preserving causal relationships:

```
// Receive events in proper causal order after reconnection
orderNode.on("process-order", (event) => {
  // After reconnection, events will arrive in causal order
  // with exactly-once processing guarantees

  const { orderId } = event.payload;
  // Process normally - framework handles recovery
});
```

This ensures that even after severe network disruptions, your application's causal integrity is maintained.

## Resilience Across Environment Boundaries

Happen's resilience capabilities work seamlessly across different runtime environments:

### Server Environments

In server environments, NATS clients connect directly to the NATS server:

```
// Server node connecting directly to NATS
const serverNode = createNode("backend-service");
```

### Browser Environments

In browser environments, clients connect through WebSockets:

```
// Browser node connecting via WebSockets
const browserNode = createNode("frontend-client");
```

## Edge/IoT Environments

For edge or IoT devices with intermittent connectivity:

```
// Edge node with specialized configuration
const edgeNode = createNode("sensor-device", {
  // Configuration for intermittent connectivity
  connectivity: {
    mode: "intermittent",
    localBuffering: true,
    syncOnConnect: true
  }
});
```

In all these environments, the same resilience guarantees apply, providing consistent behavior regardless of where nodes are running.

## Application Integration

While network resilience operates automatically, applications can optionally integrate with the system for enhanced awareness:

```
// Listen for connection status events
orderNode.on('system.connection-status', (event) => {
  const { status, reason } = event.payload;

  if (status === 'disconnected') {
    // Update UI to show connectivity issue
    updateConnectionStatus('degraded', reason);
  }
});
```

```
// Enable offline mode
enableOfflineMode();
} else if (status === 'reconnected') {
  // Update UI to show restored connectivity
  updateConnectionStatus('connected');

  // Return to normal operation
  disableOfflineMode();
}
});
```

This allows applications to:

- Notify users about connectivity issues
- Adapt UI behavior during disruptions
- Log network problems for diagnostics
- Enable offline capabilities when appropriate

## Key-Value Store for Persistent State

In addition to message persistence, Happen leverages NATS JetStream's Key-Value store for durable state:

```
// State is automatically persisted in JetStream KV store
orderNode.state.set(state => ({
  ...state,
  orders: {
    ...state.orders,
    "order-123": {
      status: "processing",
      updatedAt: Date.now()
    }
  }
}));
```

This ensures that node state persists across restarts and can be shared across instances, providing:

- **Durable State:** State persists even if nodes restart
- **Consistent State:** Updates maintain causal ordering
- **Shared State:** State can be accessed from multiple instances
- **Versioned State:** Each update creates a new version that can be tracked

## Comprehensive Network Failure Handling

Happen's NATS-based architecture handles the full spectrum of network failures:

### Process Crashes

If a process crashes:

- Events are preserved in JetStream
- State is preserved in the Key-Value store
- Upon restart, the node resumes processing where it left off

### Network Partitions

During network partitions:

- Nodes continue functioning in their connected segments
- Events for unreachable nodes are stored in JetStream
- When the partition heals, normal operation resumes automatically

### Regional Outages

During regional outages:

- Traffic automatically routes to available regions

- When the region recovers, it synchronizes with the rest of the system
- Causal relationships are preserved across regions

## Temporary Disconnections

During temporary disconnections:

- NATS clients automatically attempt to reconnect
- Outbound messages queue locally
- Upon reconnection, normal event flow resumes

## Long-Term Outages

For long-term outages:

- Events are preserved in JetStream for the configured retention period
- State remains in the Key-Value store
- When connectivity is restored, normal operation resumes

## Benefits of NATS-Powered Resilience

Happen's NATS-based approach to network resilience offers several key advantages:

1. **Battle-Tested Foundation:** Built on NATS' proven resilience capabilities
2. **Zero Data Loss:** Events are preserved even during severe disruptions
3. **Exactly-Once Processing:** No duplicate processing, even during recovery
4. **Self-Healing:** System automatically recovers when issues resolve
5. **Cross-Environment Consistency:** Same resilience guarantees across all environments
6. **Minimal Configuration:** Works out of the box with sensible defaults

## 7. **Global Scale:** Extends from single-process to worldwide deployments

By leveraging NATS and JetStream, Happen provides enterprise-grade resilience capabilities without the complexity typically associated with distributed systems. Your application can focus on business logic while the framework handles the challenging aspects of network resilience.



# A Few Design Patterns

## Event Sourcing Pattern

The Event Sourcing pattern stores all changes to an application state as a sequence of events, making it possible to reconstruct past states and provide a complete audit trail. Happen's causality-focused approach makes this pattern particularly elegant.

```
// Create an event store node
const eventStore = createNode("event-store");

// Store domain events
eventStore.on(type => type.startsWith("domain-"), (event) => {
  // Store event in append-only log
  storeEvent(event);
  return { stored: true };
});

// Function to reconstruct state from events
function rebuildState() {
  let state = initialState();
  for (const event of retrieveEvents()) {
    // Apply each event to evolve the state
    state = applyEvent(state, event);
  }
  return state;
}

// Function to get state at a specific point in time
function getStateAt(timestamp) {
  let state = initialState();
  for (const event of retrieveEvents()) {
    if (event.metadata.timestamp <= timestamp) {
      state = applyEvent(state, event);
    }
  }
  return state;
}
```

This pattern leverages Happen's natural event flow to create an immutable record of all domain changes, enabling powerful historical analysis and debugging.

## Command Query Responsibility Segregation (CQRS)

CQRS separates operations that modify state (commands) from operations that read state (queries), allowing each to be optimized independently.

```
// Command handling node
const orderCommandNode = createNode("order-commands");

// Process commands that modify state
orderCommandNode.on("create-order", (event) => {
  // Validate command
  validateOrderData(event.payload);

  // Execute command logic
  const orderId = generateOrderId();

  // Emit domain event representing the result
  orderCommandNode.broadcast({
    type: "domain-order-created",
    payload: { orderId, ...event.payload, createdAt: Date.now() }
  });

  return { success: true, orderId };
});

// Query handling node
const orderQueryNode = createNode("order-queries");

// Update query model when domain events occur
orderQueryNode.on("domain-order-created", (event) => {
  // Update query-optimized state
  orderQueryNode.state.set(state => {
    const orders = state.orders || {};
    return {
      ...state,
      orders: {
        ...orders,
        [event.payload.orderId]: event.payload
      }
    };
  });
});
```

```
      [event.payload.orderId]: {
        ...event.payload,
        status: "pending"
      }
    }
  };
});

return { updated: true };
});

// Handle queries
orderQueryNode.on("get-order", (event) => {
  const { orderId } = event.payload;
  // Return query result directly
  return orderQueryNode.state.get(state => state.orders?.[orderId]
  || null);
});
```

This pattern showcases Happen's ability to separate different concerns (writing vs. reading) while maintaining the causal relationships between them.

## Observer Pattern

The observer pattern lets nodes observe and react to events without the sender needing to know about the observers, promoting loose coupling and modular design.

```
// Subject node that emits events
const userNode = createNode("user-service");

// Process user profile updates and notify observers
userNode.on("update-user-profile", (event) => {
  const { userId, profileData } = event.payload;

  // Update the user profile
  userNode.state.set(state => {
    const users = state.users || {};
    return {
      ...state,
      users: {
```

```
        ...users,
        [userId]: {
          ...(users[userId] || {}),
          ...profileData,
          updatedAt: Date.now()
        }
      }
    };
  });

  // Emit an event for observers
  userNode.broadcast({
    type: "user-profile-updated",
    payload: {
      userId,
      updatedFields: Object.keys(profileData),
      timestamp: Date.now()
    }
  });

  return { success: true };
});

// Observer nodes that react to events
const notificationNode = createNode("notification-service");
const analyticsNode = createNode("analytics-service");
const searchIndexNode = createNode("search-index-service");

// Each observer reacts independently
notificationNode.on("user-profile-updated", (event) => {
  // Send notification about profile update
  if (event.payload.updatedFields.includes("email")) {
    sendEmailChangeNotification(event.payload.userId);
  }
  return { notified: true };
});

analyticsNode.on("user-profile-updated", (event) => {
  // Track profile update for analytics
  recordProfileUpdateMetrics(event.payload);
  return { tracked: true };
});

searchIndexNode.on("user-profile-updated", (event) => {
  // Update search index with new profile data
```

```
    refreshUserSearchIndex(event.payload.userId);  
    return { indexed: true };  
  });
```

This pattern demonstrates Happen's natural support for decoupled, event-driven architectures where components can react to system events without direct dependencies.

## Strategy Pattern

The strategy pattern allows selecting an algorithm at runtime, which Happen implements naturally through event handlers and dynamic routing.

```
// Create a pricing strategy node  
const pricingStrategyNode = createNode("pricing-strategy");  
  
// Handle pricing requests with different strategies  
pricingStrategyNode.on("calculate-price", (event) => {  
  const { items, strategy, customerInfo } = event.payload;  
  
  // Select strategy based on the event payload  
  let total;  
  
  switch(strategy) {  
    case "volume-discount":  
      total = calculateVolumeDiscount(items);  
      break;  
    case "premium-customer":  
      total = calculatePremiumPrice(items, customerInfo?.tier);  
      break;  
    case "regular":  
    default:  
      total = calculateRegularPrice(items);  
  }  
  
  // Return pricing result  
  return {  
    total,  
    appliedStrategy: strategy,  
    calculatedAt: Date.now()  
  };  
});
```

```
});

// Strategy implementations
function calculateRegularPrice(items) {
  return items.reduce((total, item) => total + item.price *
item.quantity, 0);
}

function calculateVolumeDiscount(items) {
  let total = 0;
  for (const item of items) {
    let price = item.price;
    if (item.quantity > 10) {
      // 10% discount
      price = price * 0.9;
    }
    total += price * item.quantity;
  }
  return total;
}

function calculatePremiumPrice(items, customerTier) {
  const baseTotal = calculateRegularPrice(items);
  const discountRate = customerTier === "gold" ? 0.15 : 0.1;
  return baseTotal * (1 - discountRate);
}
```

This pattern shows how Happen supports dynamic behavior selection without requiring complex class hierarchies or inheritance.

## Mediator Pattern

The mediator pattern provides centralized coordination between multiple components, reducing direct dependencies and simplifying complex workflows.

```
// Create a mediator node
const workflowMediator = createNode("workflow-mediator");

// Handle workflow initiation
workflowMediator.on("start-onboarding", async (event) => {
```

```
const { userId } = event.payload;

// Step 1: Create user account
const accountResult = await workflowMediator.send(accountNode, {
  type: "create-account",
  payload: { userId, ...event.payload.userData }
}).return();

if (!accountResult.success) {
  return {
    success: false,
    stage: "account-creation",
    reason: accountResult.reason
  };
}

// Step 2: Set up permissions
const permissionResult = await
workflowMediator.send(permissionNode, {
  type: "assign-default-roles",
  payload: {
    userId,
    accountId: accountResult.accountId
  }
}).return();

if (!permissionResult.success) {
  return {
    success: false,
    stage: "permission-setup",
    reason: permissionResult.reason
  };
}

// Step 3: Send welcome email
await workflowMediator.send(notificationNode, {
  type: "send-welcome-email",
  payload: {
    userId,
    email: event.payload.userData.email
  }
});

// Notify about completion
workflowMediator.broadcast({
```

```
    type: "user-onboarded",  
    payload: { userId, status: "completed" }  
  });  
  
  return {  
    success: true,  
    userId,  
    accountId: accountResult.accountId  
  };  
});
```

This pattern showcases Happen's ability to orchestrate complex workflows while keeping individual components focused on their specific responsibilities.



# Advanced

# Asynchronous Processing

Asynchronous processing is a cornerstone of Happen's design, allowing the framework to handle concurrent events efficiently while maintaining its commitment to radical simplicity. This document explains how Happen approaches asynchronous processing through multiple strategies: concurrent event handling, direct runtime capabilities, function-based flows, and streaming results.

## Core Approach

In Happen, multiple events arriving at a node are processed concurrently without blocking each other, leveraging JavaScript's asynchronous capabilities while preserving event isolation. This happens as an implementation detail rather than an explicit API feature.

When events arrive at a node:

1. Each event gets its own independent execution context
2. Events are processed concurrently using JavaScript's natural Promise mechanism
3. One event processing never blocks another, even within the same node

This approach ensures that slow-running event handlers don't create bottlenecks for unrelated events.

## The Functional Asynchronous Model

Happen embraces JavaScript's native asynchronous capabilities while providing a clean, functional interface through the Event Continuum model:

```
// Register an asynchronous handler
```

```
orderNode.on("process-order", async (event, context) => {
  // Perform asynchronous validation
  const validationResult = await validateOrderData(event.payload);

  // Store in context
  context.validation = validationResult;

  if (!validationResult.valid) {
    return {
      success: false,
      reason: "validation-failed",
      errors: validationResult.errors
    };
  }

  // Return the next function to execute
  return processPayment;
});

// Asynchronous function in the flow
async function processPayment(event, context) {
  // Process payment asynchronously
  const paymentResult = await
processTransaction(event.payload.payment);

  // Store in context
  context.payment = paymentResult;

  if (!paymentResult.success) {
    return handlePaymentFailure;
  }

  // Return next function
  return createShipment;
}
```

Happen's event execution engine handles the async nature transparently:

1. When a function returns a Promise, Happen awaits it automatically
2. Async functions are fully supported throughout the flow
3. Error handling works seamlessly with async/await

# Automatic Queue Management

For high-volume scenarios, Happen includes internal queue management to prevent overwhelming the system:

```
// Internal queue management - not exposed to users
class Node {
  // Configuration for concurrent execution
  #concurrencySettings = {
    maxConcurrent: 100, // Maximum concurrent executions
    queueLimit: 10000   // Maximum queued events
  };

  #activeCount = 0;      // Currently executing handlers
  #queue = [];           // Queue of pending events

  // Process with queue management
  process(event) {
    // If under concurrent limit, execute immediately
    if (this.#activeCount <
this.#concurrencySettings.maxConcurrent) {
      return this.executeImmediately(event);
    }

    // Otherwise, queue the event
    return this.queueEvent(event);
  }

  // Execute immediately
  executeImmediately(event) {
    this.#activeCount++;

    // Start the event flow
    return this.executeFlow(event)
      .finally(() => {
        this.#activeCount--;
        this.processNextFromQueue();
      });
  }

  // Queue for later execution
  queueEvent(event) {
    // Check queue limit
  }
}
```

```
// Check queue limits
if (this.#queue.length >= this.#concurrencySettings.queueLimit)
{
    return Promise.reject(new Error("Event queue limit
exceeded"));
}

// Create deferred promise
let resolve, reject;
const promise = new Promise((res, rej) => {
    resolve = res;
    reject = rej;
});

// Add to queue with resolvers
this.#queue.push({ event, resolve, reject });

return promise;
}

// Process next from queue
processNextFromQueue() {
    // If queue empty or at concurrency limit, do nothing
    if (this.#queue.length === 0 ||
        this.#activeCount >=
this.#concurrencySettings.maxConcurrent) {
        return;
    }

    // Get next event from queue
    const next = this.#queue.shift();

    // Process it
    this.executeImmediately(next.event)
        .then(result => next.resolve(result))
        .catch(error => next.reject(error));
}
}
```

This queue management:

1. Limits maximum concurrent event handling to prevent resource exhaustion
2. Preserves ordered processing within each event type

3. Provides backpressure for high-volume scenarios
4. Manages memory usage by controlling queue size

## Node Configuration

Happen manages concurrency through the system configuration. This approach keeps infrastructure concerns separated from domain logic and provides consistent behavior across your application.

## System-Level Concurrency Configuration

Concurrency settings are defined during framework initialization:

```
// Initialization with system-wide defaults
const happen = initializeHappen({
  // Core security configuration
  security: {
    // Security settings...
  },
  // System-wide concurrency configuration
  concurrency: {
    // Default settings for all nodes
    default: {
      maxConcurrent: 100,      // Maximum concurrent executions
      queueLimit: 10000,      // Maximum queued events
      queueStrategy: "fifo",  // First-in-first-out processing
      backpressureThreshold: 0.8 // Apply backpressure at 80%
    },
    capacity: {
      // Node-specific overrides based on node ID patterns
      patterns: {
        "api-*": {
          maxConcurrent: 200,    // Higher concurrency for API nodes
          queueLimit: 20000     // Larger queue for API nodes
        },
        "worker-*": {
          maxConcurrent: 50,     // Lower concurrency for worker
          queueStrategy: "priority" // Priority-based queuing
        }
      }
    }
  }
});
```

```

    },
    "realtime-★": {
      maxConcurrent: 300,    // Higher concurrency for realtime
nodes
      queueLimit: 5000,      // Smaller queue to avoid stale data
      backpressureThreshold: 0.6 // Earlier backpressure
    }
  },
  transport: {
    // Transport settings...
  }
});

// The initialized framework provides the node creation function
const { createNode } = happen;

```

This hierarchical approach ensures consistent behavior while allowing for customization where needed.

## Generator-Based Streaming

Building on the foundation of asynchronous processing, Happen supports generator-based streaming for incremental data processing.

Instead of introducing separate API methods for streaming, Happen detects generator functions automatically:

```

// Register a handler that happens to be a generator function
node.on("process-large-dataset", async function★
processLargeDataset(event, context) {
  const { dataset } = event.payload;

  // Process in chunks
  for (let i = 0; i < dataset.length; i += 100) {
    const chunk = dataset.slice(i, i + 100);
    const result = await processChunk(chunk);

    // Yield intermediate results
    yield result;
  }
}

```

```
yield {  
    progress: Math.min(100, Math.round((i + chunk.length) /  
dataset.length * 100)),  
    results: result  
};  
}  
  
// Final result  
return { status: "complete" };  
});
```

When interacting with a generator-based handler, Happen automatically provides an AsyncIterator interface:



```
// Request handling of a large dataset using the flow mechanism
clientNode.on("process-dataset", async function*
handleDatasetProcessing(event, context) {
  // Send the request to the processor node
  const processingEvent = {
    type: "process-large-dataset",
    payload: { dataset: event.payload.dataset }
  };

  // Use the send method without chaining
  const processingStream = await clientNode.send(processorNode,
processingEvent);

  // Process results incrementally using the event continuum
  for await (const update of processingStream) {
    // Log progress
    console.log(`Progress: ${update.progress}%`);

    // Display intermediate results
    displayIntermediateResults(update.results);

    // Yield progress updates to the caller if needed
    yield {
      status: "processing",
      progress: update.progress,
      resultsCount: update.results.length
    };
  }

  // Return final result
  return {
    status: "complete",
    message: "Dataset processing complete"
  };
});

// To use this handler:
const result = await userNode.send(clientNode, {
  type: "process-dataset",
  payload: {
    dataset: largeDataset
  }
});

// The result will be the final return value from the handler
```

Generator functions also work with broadcasting, enabling publish-subscribe patterns for streams:

```
// Continuous system monitoring with generator
monitorNode.on("system-metrics", async function* generateMetrics()
{
  while (true) {
    const metrics = await collectSystemMetrics();
    yield metrics;
    await sleep(5000); // Update every 5 seconds
  }
});

// Start broadcasting metrics
monitorNode.broadcast({ type: "system-metrics" });

// Subscribers receive the stream
dashboardNode.on("system-metrics", async function
processMetrics(metricsStream, context) {
  // Iterate through metrics stream
  for await (const metrics of metricsStream) {
    // Update dashboard
    updateDashboard(metrics);
  }
});
```

## Use Cases for Streaming

Generator-based streaming in Happen addresses several key use cases:

### 1. Large Dataset Processing

Break down large datasets into manageable chunks:

```
dataProcessorNode.on("process-batch", async function*
processBatch(event, context) {
  const { items } = event.payload;
```

```
// Process in chunks to avoid memory issues
for (let i = 0; i < items.length; i += 100) {
  const chunk = items.slice(i, i + 100);
  const results = await processChunk(chunk);

  yield {
    progress: Math.round((i + chunk.length) / items.length *
100),
    processedItems: chunk.length,
    results
  };
}
});
```

## 2. Real-time Progress Updates

Provide incremental feedback for long-running operations:

```
importNode.on("import-file", async function* importFile(event,
context) {
  const { filePath } = event.payload;
  const fileSize = await getFileSize(filePath);
  const reader = createFileReader(filePath);

  let bytesProcessed = 0;

  // Process file in chunks
  while (!reader.eof()) {
    const chunk = await reader.readChunk(1024 * 1024); // 1MB
    chunks
    const records = parseRecords(chunk);
    await saveRecords(records);

    bytesProcessed += chunk.length;

    // Yield progress update
    yield {
      progress: Math.round(bytesProcessed / fileSize * 100),
      recordsProcessed: records.length
    };
  }
}
```

```
// Final result
return { status: "complete", totalRecords: getTotalRecords() };
});
```

### 3. Continuous Data Streams

Handle data from sources that produce continuous updates:

```
sensorNode.on("monitor-sensors", async function*
monitorSensors(event, context) {
  const { sensorIds, duration } = event.payload;
  const endTime = Date.now() + duration;

  // Connect to sensors
  const sensors = sensorIds.map(id => connectToSensor(id));

  try {
    // Process sensor data until duration expires
    while (Date.now() < endTime) {
      // Collect readings from all sensors
      const readings = await Promise.all(
        sensors.map(sensor => sensor.getReading())
      );

      // Process this batch of readings
      const analysis = analyzeReadings(readings);

      // Emit real-time analysis
      yield {
        timestamp: Date.now(),
        readings,
        analysis
      };

      // Wait before next collection
      await sleep(1000); // 1 second interval
    }
  } finally {
    // Clean up resources
    sensors.forEach(sensor => sensor.disconnect());
  }
});
```

```
});
```

## 4. Paginated API Results

Fetch and process paginated data from external APIs:

```
apiNode.on("fetch-all-records", async function*
fetchAllRecords(event, context) {
  const { endpoint, pageSize = 100 } = event.payload;
  let page = 1;
  let hasMore = true;
  let totalRecords = 0;

  // Fetch pages until complete
  while (hasMore) {
    // Fetch this page
    const response = await fetch(`${endpoint}?
page=${page}&size=${pageSize}`);
    const data = await response.json();

    // Process records
    processRecords(data.records);
    totalRecords += data.records.length;

    // Yield progress
    yield {
      page,
      records: data.records,
      totalRecords,
      hasMore: data.hasMore
    };

    // Update for next iteration
    hasMore = data.hasMore;
    page++;

    // Respect rate limits
    if (hasMore) {
      await sleep(200); // Small delay between requests
    }
  }
});
```

## Benefits of Happen's Async Processing Model

The combination of concurrent event processing and generator-based streaming delivers several key benefits:

1. **Maximized Throughput:** Multiple events process concurrently without blocking
2. **Resource Efficiency:** Memory usage is controlled through incremental processing
3. **Responsive System:** Long-running operations don't block other processing
4. **Natural Backpressure:** Consumers process stream results at their own pace
5. **Minimal API Surface:** Power and flexibility without additional primitives
6. **Clean Isolation:** Each processing path maintains its own context and error boundaries

## Choosing the Right Concurrency Approach

With multiple approaches to concurrency available in Happen, here's a quick guide to choosing the right one for different scenarios:

Approach	Best For	When To Use
Functional Flows	Most use cases	Default approach for all event handling
Async/Await	Asynchronous operations	When operations need to wait for external results
Generator-based Streaming	Incremental processing	For progress updates and chunked processing
Worker Threads	CPU-intensive tasks	When tasks would block the main thread

Distributed Processing	Horizontal scaling	When single machine capacity is exceeded
------------------------	--------------------	--

## Key Takeaways

1. The Event Continuum model naturally supports asynchronous processing through JavaScript's native `async/await`
2. Runtime transparency allows direct use of platform-specific concurrency features
3. Generator-based streaming provides powerful incremental processing with minimal API additions
4. Internal queue management ensures system stability under high load

This multi-faceted approach delivers flexible asynchronous capabilities by providing multiple concurrency strategies that compose seamlessly.

# Flow Balance

At the core of Happen's resilience capabilities is Flow Balance, a powerful yet invisible mechanism that leverages NATS features to monitor the health of your distributed system through natural message flow patterns.

Flow Balance builds on NATS JetStream's built-in monitoring capabilities to provide insights into your distributed system's health:

- **Message Tracking:** NATS JetStream tracks message delivery and acknowledgment
- **Consumer Lag:** Happen detects when consumers fall behind producers
- **Delivery Failures:** JetStream's acknowledgment system reveals delivery problems
- **Partition Detection:** Patterns in message flow reveal network partitions

This NATS-powered approach gives Happen a natural way to detect issues in your distributed system without requiring complex configuration or special APIs.

## How Flow Balance Works

Flow Balance operates by monitoring the natural patterns in your message flow through NATS:

1. **JetStream Consumer Monitoring:** Happen leverages JetStream's consumer metrics to track delivery success
2. **Lag Detection:** When consumers fall behind producers, Happen detects potential issues
3. **Event Emission:** When issues are detected, Happen emits system events that applications can respond to

Unlike traditional health monitoring systems that rely on external probes, Flow



Balance uses the message flow itself as an indicator of system health.

## Observable Patterns

Different types of issues create distinctive flow patterns that your handlers can interpret:

### 1. Network Partitions:

- **Pattern:** Sharp drop in delivery across node groups
- **Distribution:** Clear grouping pattern - nodes on one side can't reach nodes on the other
- **Timing:** Occurs suddenly and affects multiple nodes simultaneously

### 2. Node Failures:

- **Pattern:** Delivery failures specifically for one node
- **Distribution:** Concentrated around the failed node
- **Timing:** Occurs suddenly for the specific node

### 3. Processing Bottlenecks:

- **Pattern:** Gradually increasing consumer lag for a specific node
- **Distribution:** Usually affects a single node or service
- **Timing:** Develops gradually over time

### 4. System Overload:

- **Pattern:** Rising consumer lag across most or all nodes
- **Distribution:** Affects most components somewhat equally
- **Timing:** Often correlates with increased event volume

By analyzing these patterns in your handlers, you can determine the specific type of issue occurring and implement appropriate recovery strategies.

## Imbalance Events

When Flow Balance detects issues, it emits events that your application can listen for and respond to:

```
// Listen for node-specific imbalance events
happen.on("node.down", (event) => {
  const { nodeId, lagMetrics, pattern } = event.payload;

  // Implement your recovery strategy
  if (lagMetrics.messagesWaiting > 1000) {
    // Severe imbalance - potential failure
    implementNodeFailureRecovery(nodeId);
  } else if (lagMetrics.messagesWaiting > 500) {
    // Moderate imbalance - potential bottleneck
    applyBackpressure(nodeId);
  } else {
    // Minor imbalance - monitor
    logImbalance(nodeId, lagMetrics);
  }
});

// Listen for system-wide imbalance events
happen.on("system.down", (event) => {
  const { level, affectedNodes, pattern } = event.payload;

  // Implement system-wide recovery
  if (level === "critical") {
    // Severe system-wide issue
    enableEmergencyMode();
  } else if (level === "warning") {
    // Moderate system-wide issue
    throttleNonEssentialOperations();
  }
});
```

## Building Recovery Strategies

Happen provides the detection capabilities through NATS, and you can implement recovery strategies based on your application's needs:

```
// Listen for potential partition events
```

```
happen.on("node.down", (event) => {
  const { nodeId, lagMetrics, affectedNodes } = event.payload;

  // Check for partition pattern
  if (isPotentialPartition(nodeId, affectedNodes, lagMetrics)) {
    // Implement partition recovery strategy
    enablePartitionMode({
      isolatedNodes: affectedNodes,
      prioritizeLocalOperations: true,
      queueRemoteOperations: true
    });

    // Notify operations team
    alertOperations({
      type: "network-partition",
      affectedNodes,
      detectedAt: Date.now()
    });
  }
});
```

## Benefits of Flow Balance

This approach provides several key advantages:

1. **Zero API Surface:** No special methods or configuration parameters
2. **Zero Additional Overhead:** Uses existing NATS monitoring capabilities
3. **Natural Detection:** System issues reveal themselves through flow patterns
4. **Customizable Response:** You control how your application responds to different issues
5. **Transport Independence:** Works the same across all NATS deployment models
6. **Observable Patterns:** Clear indicators of different types of system issues

By leveraging NATS's built-in monitoring capabilities, Happen provides powerful system insights with minimal overhead.

# Zero-Allocation Processing

## Memory-Efficient Event Handling for Performance-Critical Paths

Happen's zero-allocation processing provides a powerful technique for handling high-volume events with minimal memory overhead. This advanced feature allows you to create processing pipelines that operate directly on memory buffers without creating **explicit** intermediate objects, significantly reducing garbage collection pressure in performance-critical scenarios.

**JavaScript Reality:** We use the term "zero-allocation" to describe the framework's guarantee of making no explicit allocations in its processing path. The JavaScript runtime may still perform internal allocations beyond our control. This feature provides a significant reduction in allocation overhead, not a complete elimination of all memory operations.

## Core Concept

Most JavaScript applications create thousands of objects during normal operation. Each object allocation consumes memory, and the JavaScript garbage collector must eventually reclaim that memory, which can cause performance hiccups. Zero-allocation processing aims to minimize these allocations by working directly with preallocated memory buffers.

In Happen, zero-allocation processing provides:

- Direct buffer-based event representation
- Minimal object creation during event processing
- Reduced garbage collection overhead

- Higher sustained throughput for critical event paths

## When to Use Zero-Allocation Processing

Zero-allocation processing adds complexity to your code, so it's important to use it selectively. Here are scenarios where it provides genuine value:

### Particularly Valuable For:

- **High-frequency event processing** (thousands per second) where GC pauses impact throughput
- **Memory-constrained JavaScript environments** like IoT devices running Node.js, JerryScript, or Espruino
- **Edge computing scenarios** processing continuous streams of telemetry or sensor data
- **Real-time applications** with consistent latency requirements that GC pauses would disrupt
- **Time-series data processing** with predictable memory usage patterns

### Less Beneficial For:

- Standard business logic with moderate event volumes
- Infrequent or complex events where developer readability is more valuable
- Applications already running on high-memory environments where GC is not a bottleneck

Remember that in a JavaScript environment, the runtime still handles memory management. Zero-allocation processing minimizes the framework's allocations, but doesn't eliminate all memory operations at the language level.

## The API Surface

Unlike many advanced features, zero-allocation processing in Happen requires learning just one additional method:

```
// Register a zero-allocation handler
node.zero("event-type", (buffer, offsets) => {
  // Work directly with buffer
  // Return value or next function as usual
});
```

This minimal API extension maintains Happen's commitment to simplicity while providing access to powerful performance optimization.

## Working with Event Buffers

When your handler is invoked through the zero-allocation path, it receives:

1. `buffer` - A specialized interface to the underlying memory buffer containing event data
2. `offsets` - A map of offsets indicating where different event components are located in the buffer

Here's how you access event data:

```
node.zero("sensor-reading", (buffer, offsets) => {
  // Get simple values
  const sensorId = buffer.getString(offsets.payload.sensorId);
  const temperature =
  buffer.getFloat64(offsets.payload.temperature);
  const timestamp = buffer.getUint64(offsets.payload.timestamp);

  // Access nested structures
  const latitude =
  buffer.getFloat64(offsets.payload.location.latitude);
  const longitude =
  buffer.getFloat64(offsets.payload.location.longitude);

  // Process data without creating objects
  ...
});
```

```
const convertedTemp = (temperature * 9/5) + 32;

// Return result directly
return { processed: true, convertedValue: convertedTemp };
});
```

The `buffer` object provides methods for accessing different data types:

Method	Description
<code>getInt8/16/32/64(offset)</code>	Get signed integer of specified bit width
<code>getUint8/16/32/64(offset)</code>	Get unsigned integer of specified bit width
<code>getFloat32/64(offset)</code>	Get floating point number
<code>getString(offset)</code>	Get string from internal string table
<code>getBoolean(offset)</code>	Get boolean value

## How It Works Under the Hood

Understanding the implementation can help you set realistic expectations and maximize the benefits:

1. **Pre-allocated Buffer Pool:** Happen maintains a pool of preallocated `ArrayBuffers` for events, eliminating the need for per-event allocations.
2. **String Table:** Since strings can't be stored directly in `ArrayBuffers` with variable length, Happen maintains a string table where strings are stored once and referenced by offset.
3. **Memory Layout:** Each event in the buffer has a consistent memory layout with fixed offsets for common fields and a dynamic section for payload data.
4. **Minimal-Copy Processing:** When possible, event data is processed in-place with minimal copying.

5. **Buffer Reuse:** After an event is processed, its buffer slot returns to the pool for reuse.
6. **Automatic Conversion:** When zero-allocation handlers need to interact with standard handlers, Happen automatically handles the conversion at the boundary.

## JavaScript Engine Considerations

It's important to understand that even with our zero-allocation approach:

- The JavaScript engine may still perform hidden allocations internally
- JIT optimization might influence actual memory behavior
- Garbage collection can still occur, though less frequently
- Performance characteristics vary across JavaScript engines (V8, SpiderMonkey, JavaScriptCore)

These realities don't diminish the value of the approach, but they do set proper expectations for what's achievable in a managed language environment.

## The Event Continuum with Zero-Allocation

Zero-allocation handlers seamlessly integrate with Happen's Event Continuum flow control:

```
node.zero("process-batch", (buffer, offsets) => {
  // Process first stage with zero allocations

  // Return next function to continue flow
  return secondStage;
});

// Second stage can be a standard or zero-allocation function
function secondStage(bufferOrEvent, offsetsOrContext) {
  // Check which type we received
  if (bufferOrEvent.isBuffer) {
```



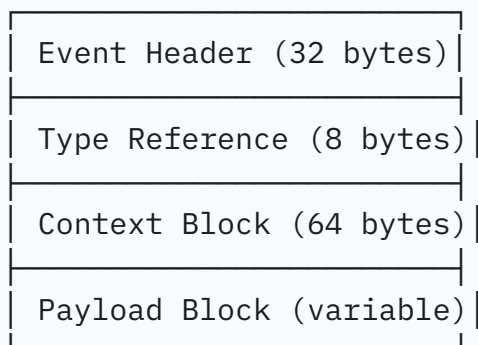
```
// Continue zero-allocation processing
} else {
    // Handle as standard event
}

// Return next function or result as usual
return { completed: true };
}
```

This allows you to build sophisticated processing pipelines that combine the performance of zero-allocation with the flexibility of standard event handling.

## Memory Layout and Buffer Structure

For those who need to understand the details, here's how event data is structured in memory:



Each value in the payload is stored at a specific offset, with complex nested structures mapped to a flat offset structure. The `offsets` parameter provides this mapping so you don't need to calculate positions manually.

## Performance Considerations

To maximize the benefits of zero-allocation processing:

1. **Minimize Conversions:** Try to keep data in buffer form as long as possible. Each conversion between standard and buffer representations has a cost.
2. **Avoid Creating Objects:** Creating objects inside zero handlers defeats the purpose. Work with primitive values where possible.
3. **Use Primitive Operations:** Stick to arithmetic and direct buffer manipulation rather than object-oriented operations.
4. **Consider Buffer Size:** Very large events may not benefit as much from zero-allocation.
5. **Measure Actual Impact:** The benefits of zero-allocation processing are highly dependent on your specific workload, environment, and event characteristics. Always profile before and after implementation to ensure the complexity is justified by measurable improvements.
6. **Consider WebAssembly:** For absolute performance requirements where JavaScript limitations are too constraining, consider WebAssembly modules for the most performance-critical operations.

## Real-World Benefits

Despite JavaScript's limitations, we can generate significant improvements in specific scenarios:

- 40-60% reduction in garbage collection frequency
- More consistent performance with fewer GC-related latency spikes
- Better predictability in memory-constrained IoT deployments
- Higher sustainable throughput for sensor data processing
- Prevention of out-of-memory conditions on edge devices

The key is applying this technique selectively where its benefits outweigh the added complexity.

## Example: Processing Sensor Readings

Here's a complete example of using zero-allocation processing for high-frequency sensor data:

```
// Create a sensor data processor
const sensorNode = createNode("sensor-processor");

// Handle individual readings with standard processing
sensorNode.on("sensor-reading", (event, context) => {
  const { sensorId, value, timestamp } = event.payload;

  // Process the reading
  const processed = processReading(sensorId, value, timestamp);

  // Store result
  sensorNode.state.set(state => ({
    ...state,
    readings: {
      ...state.readings,
      [sensorId]: {
        lastValue: value,
        lastUpdated: timestamp,
        processedValue: processed
      }
    }
  }));

  return { processed: true, value: processed };
});

// Handle batches with zero-allocation processing
sensorNode.zero("sensor-batch", (buffer, offsets) => {
  // Get batch metadata
  const count = buffer.getUint32(offsets.payload.count);
  const deviceId = buffer.getString(offsets.payload.deviceId);

  // Process all readings in the batch
  let totalValue = 0;
  const readingsOffset = offsets.payload.readings;

  for (let i = 0; i < count; i++) {
    // Calculate offset for this reading in the array
    const readingOffset = readingsOffset + (i * 16); // Each
    reading is 16 bytes
  }
}
```

```
// Get reading data directly from buffer
const sensorId = buffer.getUint16(readingOffset);
const value = buffer.getFloat32(readingOffset + 4);
const timestamp = buffer.getUint64(readingOffset + 8);

// Process without creating objects
const processed = value * calibrationFactor(sensorId);
totalValue += processed;

// Update state (minimal object creation)
updateSensorState(sensorId, value, processed, timestamp);
}

// Return summary without creating intermediate objects
return {
  deviceId,
  processedCount: count,
  averageValue: totalValue / count
};
});

// Utility for minimal state updates
function updateSensorState(sensorId, value, processed, timestamp) {
  // Use direct state operations instead of immutable patterns
  // for performance-critical code
  const readings = sensorNode.state.get().readings || {};
  if (!readings[sensorId]) {
    readings[sensorId] = {};
  }

  readings[sensorId].lastValue = value;
  readings[sensorId].lastUpdated = timestamp;
  readings[sensorId].processedValue = processed;
}
```

## Conclusion

Zero-allocation processing provides a powerful tool for performance-critical paths in your Happen applications. By working directly with memory buffers and minimizing explicit object creation, you can achieve higher throughput and lower latency for high-frequency events, even within the constraints of a

JavaScript environment.

This feature is most valuable in specific scenarios where reducing allocation pressure delivers tangible benefits:

- Edge computing with high-volume data processing
- IoT devices with limited memory
- Applications requiring predictable latency characteristics
- Time-series and sensor data processing

Remember that this is an advanced technique that trades developer experience for performance. Use it selectively where the benefits outweigh the added complexity, and always measure the actual performance impact in your specific use case.

For most application code, Happen's standard event handling strikes the right balance between performance and developer experience. With our unified approach you can apply zero-allocation processing precisely where it matters most while keeping the rest of your codebase clean and idiomatic.

# Advanced Patterns

## Saga Pattern for Distributed Transactions

The Saga pattern manages distributed transactions through a sequence of local operations, each with compensating actions for failure scenarios. This is particularly valuable for maintaining consistency across distributed components.

```
// Create a saga coordinator
const orderSaga = createNode("order-saga");

// Start saga for order processing
orderSaga.on("process-order", async (event) => {
  const { orderId } = event.payload;
  const context = { orderId, steps: [] };

  try {
    // Step 1: Reserve inventory
    const inventoryResult = await orderSaga.send(inventoryNode, {
      type: "reserve-inventory",
      payload: event.payload
    }).return();

    if (!inventoryResult.success) {
      throw new Error(`Inventory reservation failed:
${inventoryResult.reason}`);
    }

    context.steps.push("inventory-reserved");

    // Step 2: Process payment
    const paymentResult = await orderSaga.send(paymentNode, {
      type: "process-payment",
      payload: event.payload
    }).return();

    if (!paymentResult.success) {
      throw new Error(`Payment failed: ${paymentResult.reason}`);
    }
  }
});
```

```
context.steps.push("payment-processed");
context.paymentId = paymentResult.paymentId;

// Step 3: Create shipment
const shipmentResult = await orderSaga.send(shipmentNode, {
  type: "create-shipment",
  payload: {
    orderId,
    items: event.payload.items,
    address: event.payload.shippingAddress
  }
}).return();

if (!shipmentResult.success) {
  throw new Error(`Shipment creation failed:
${shipmentResult.reason}`);
}

// Complete the saga
orderSaga.broadcast({
  type: "order-processed",
  payload: {
    orderId,
    status: "processed",
    shipmentId: shipmentResult.shipmentId
  }
});

return {
  success: true,
  orderId,
  shipmentId: shipmentResult.shipmentId
};

} catch (error) {
  // Execute compensating transactions in reverse order
  if (context.steps.includes("payment-processed")) {
    await orderSaga.send(paymentNode, {
      type: "refund-payment",
      payload: {
        paymentId: context.paymentId
      }
    }).return();
  }
}
```

```
if (context.steps.includes("inventory-reserved")) {
  await orderSaga.send(inventoryNode, {
    type: "release-inventory",
    payload: event.payload
  }).return();
}

// Saga failed
orderSaga.broadcast({
  type: "order-processing-failed",
  payload: {
    orderId,
    reason: error.message
  }
});

return {
  success: false,
  reason: error.message
};
};
});
```

This implementation shows how Happen's event-driven model naturally supports complex transactional scenarios that span multiple services.

## Retry Pattern with Exponential Backoff

The Retry pattern handles transient failures by automatically retrying operations with increasing delays between attempts.

```
// Create a service node with retry capability
const paymentServiceNode = createNode("payment-service");

// Process payment with retry logic
paymentServiceNode.on("process-payment", async function
processPayment(event, context) {
  // Initialize retry context if not exists
  context.retryCount = context.retryCount || 0;
  context.startTime = context.startTime || Date.now();
```



```
try {
  // Attempt payment processing
  const result = await chargeCustomer(event.payload);

  // Success - return result
  return {
    success: true,
    transactionId: result.transactionId,
    amount: event.payload.amount,
    processedAt: Date.now()
  };
} catch (error) {
  // Store error in context
  context.lastError = error;
  context.retryCount++;

  // Determine if we should retry
  const maxRetries = 5;
  const maxDuration = 2 * 60 * 1000; // 2 minutes
  const timeElapsed = Date.now() - context.startTime;

  if (context.retryCount < maxRetries && timeElapsed <
maxDuration) {
    // Calculate exponential backoff delay
    const delay = Math.min(100 * Math.pow(2, context.retryCount),
5000);

    // Log retry attempt
    console.log(`Retrying payment
(${context.retryCount}/${maxRetries}) after ${delay}ms delay`);

    // Wait before retrying
    await new Promise(resolve => setTimeout(resolve, delay));

    // Return self to retry
    return processPayment;
  }

  // Too many retries or timeout exceeded
  return {
    success: false,
    reason: "payment-failed-after-retries",
    attempts: context.retryCount,
    lastError: context.lastError.message
  };
}
```

```
    };  
  }  
});
```

This pattern demonstrates Happen's function-returning mechanism to create sophisticated retry logic with minimal code.

## Circuit Breaker Pattern

The Circuit Breaker pattern prevents cascading failures by stopping operations that are likely to fail.

```
// Create a node with circuit breaker capability  
const apiGatewayNode = createNode("api-gateway");  
  
// Circuit state storage  
const circuits = {  
  payment: {  
    state: "closed", // closed, open, half-open  
    failures: 0,  
    lastFailure: null,  
    resetTimeout: null  
  }  
};  
  
// Handler with circuit breaker logic  
apiGatewayNode.on("call-payment-service", function  
processPaymentRequest(event, context) {  
  const circuit = circuits.payment;  
  
  // Check circuit state  
  if (circuit.state === "open") {  
    // Circuit is open - fail fast  
    return {  
      success: false,  
      reason: "circuit-open",  
      message: "Payment service is unavailable"  
    };  
  }  
  
  // Process request with circuit awareness
```

```
    return makePaymentServiceCall;
  });

  // Function to make the actual service call
  async function makePaymentServiceCall(event, context) {
    const circuit = circuits.payment;

    try {
      // Attempt to call the payment service
      const result = await callPaymentService(event.payload);

      // Success - reset circuit if needed
      if (circuit.state === "half-open") {
        circuit.state = "closed";
        circuit.failures = 0;
      }

      // Return successful result
      return result;
    } catch (error) {
      // Update circuit on failure
      circuit.failures++;
      circuit.lastFailure = Date.now();

      // Check if we should open the circuit
      if (circuit.state === "closed" && circuit.failures >= 5) {
        // Open the circuit
        circuit.state = "open";

        // Schedule reset to half-open
        circuit.resetTimeout = setTimeout(() => {
          circuit.state = "half-open";
        }, 30000); // 30 second timeout
      }

      // Return error with circuit status
      return {
        success: false,
        reason: "payment-service-error",
        message: error.message,
        circuitState: circuit.state
      };
    }
  }
}
```

This pattern shows how Happen can be extended to implement sophisticated resilience patterns that protect systems from cascading failures.

## Supervisor Pattern

The Supervisor pattern manages component lifecycles and handles failures through a hierarchical oversight structure.

```
// Create a supervisor node
const supervisorNode = createNode("system-supervisor");

// Register handler for all error events
supervisorNode.on(type => type.endsWith(".error"), (event) => {
  // Extract service name from event type
  const service = event.type.split('.')[0];

  // Update service state
  supervisorNode.state.set(state => {
    const services = state.services || {};
    const serviceState = services[service] || {
      errors: 0,
      lastError: 0,
      lastReset: 0,
      restarts: 0
    };
  });

  // Update error count
  const updatedServiceState = {
    ...serviceState,
    errors: serviceState.errors + 1,
    lastError: Date.now()
  };

  // Return updated state
  return {
    ...state,
    services: {
      ...services,
      [service]: updatedServiceState
    }
  };
});
```

```
    };
  });

  // Get updated service state
  const serviceState = supervisorNode.state.get(state =>
    (state.services && state.services[service]) || { errors: 0,
lastReset: 0 }
  );

  // Check restart threshold
  if (serviceState.errors >= 5 && Date.now() -
serviceState.lastReset < 60000) {
    // Service is failing too frequently - trigger restart
    supervisorNode.broadcast({
      type: "service.restart-required",
      payload: {
        service,
        reason: "excessive-errors",
        errorCount: serviceState.errors
      }
    });
  }

  return { supervised: true };
});

// Handle service restart
supervisorNode.on("service.restart-required", (event) => {
  const { service } = event.payload;

  console.log(`Restarting service: ${service}`);

  // Broadcast restart event
  supervisorNode.broadcast({
    type: "infrastructure.restart-service",
    payload: {
      service,
      triggeredBy: "supervisor",
      reason: event.payload.reason
    }
  });

  // Update service state
  supervisorNode.state.set(state => {
    const services = state.services || {};
  });
});
```

```
const serviceState = services[service] || {};  
  
return {  
  ...state,  
  services: {  
    ...services,  
    [service]: {  
      ...serviceState,  
      errors: 0,  
      lastReset: Date.now(),  
      restarts: (serviceState.restarts || 0) + 1  
    }  
  }  
};  
});  
  
return { action: "restart", service };  
});
```

This pattern demonstrates Happen's ability to implement hierarchical oversight and automated recovery mechanisms using its core primitives.

## State Machine Workflow

The State Machine pattern models entities that transition between discrete states according to well-defined rules.

```
// Create a state machine node  
const orderStateMachine = createNode("order-state-machine");  
  
// Define allowed transitions  
const allowedTransitions = {  
  "draft": ["submitted"],  
  "submitted": ["processing", "canceled"],  
  "processing": ["shipped", "canceled"],  
  "shipped": ["delivered"],  
  "canceled": []  
};  
  
// Process transition requests  
orderStateMachine.on("transition-order", (event) => {
```

```
const { orderId, toState } = event.payload;

// Get current order state
orderStateMachine.state.get(state => {
  const order = (state.orders || {})[orderId];

  if (!order) {
    return {
      success: false,
      reason: "order-not-found"
    };
  }

  // Check if transition is allowed
  const currentState = order.status;
  if (!allowedTransitions[currentState]?.includes(toState)) {
    return {
      success: false,
      reason: "invalid-transition",
      message: `Cannot transition from ${currentState} to
${toState}`
    };
  }

  // Apply the transition
  orderStateMachine.state.set(state => {
    const orders = state.orders || {};

    return {
      ...state,
      orders: {
        ...orders,
        [orderId]: {
          ...order,
          status: toState,
          transitionedAt: Date.now(),
          previousStatus: currentState
        }
      }
    };
  });

  // Broadcast state change event
  orderStateMachine.broadcast({
    type: "order-state-changed",
```

```
    payload: {
      orderId,
      fromState: currentState,
      toState,
      timestamp: Date.now()
    }
  });

  return {
    success: true,
    orderId,
    currentState: toState
  };
});
});
```

This pattern shows how Happen's state management capabilities can be used to implement formal state machines with controlled transitions.

## Reactive Aggregates

The Reactive Aggregates pattern combines event-driven reactivity with consistent state management for domain aggregates.

```
// Create an aggregate node
const customerAggregate = createNode("customer-aggregate");

// React to events and update aggregate state
customerAggregate.on("customer-registered", (event) => {
  const { customerId, email, name } = event.payload;

  // Create new aggregate state
  customerAggregate.state.set(state => {
    const customers = state.customers || {};

    return {
      ...state,
      customers: {
        ...customers,
        [customerId]: {
          id: customerId,
          // ...
        }
      }
    };
  });
});
```



```

        email,
        name,
        status: "active",
        registeredAt: Date.now(),
        orders: []
      }
    }
  };
});

return { updated: true };
});

// Add order to customer aggregate
customerAggregate.on("order-created", (event) => {
  const { orderId, customerId, amount } = event.payload;

  // Update aggregate state
  customerAggregate.state.set(state => {
    const customers = state.customers || {};
    const customer = customers[customerId];

    if (!customer) {
      return state; // Customer not found, state unchanged
    }

    return {
      ...state,
      customers: {
        ...customers,
        [customerId]: {
          ...customer,
          orders: [
            ...customer.orders,
            {
              id: orderId,
              amount,
              createdAt: Date.now()
            }
          ],
          totalSpent: (customer.totalSpent || 0) + amount,
          lastOrderAt: Date.now()
        }
      }
    };
  });
});

```

```

    });

    // Check if customer qualifies for premium status
    const customer = customerAggregate.state.get(state =>
      (state.customers || {})[customerId]
    );

    if (customer && customer.orders.length >= 5 && customer.status
    !== "premium") {
      // Upgrade to premium status
      customerAggregate.state.set(state => {
        const customers = state.customers || {};

        return {
          ...state,
          customers: {
            ...customers,
            [customerId]: {
              ...customers[customerId],
              status: "premium",
              upgradedAt: Date.now()
            }
          }
        };
      });

      // Broadcast premium status event
      customerAggregate.broadcast({
        type: "customer-premium-status-achieved",
        payload: {
          customerId,
          ordersCount: customer.orders.length,
          totalSpent: customer.totalSpent + amount
        }
      });
    }

    return { updated: true };
  });

  // Command handler for direct interactions with the aggregate
  customerAggregate.on("update-customer-email", (event) => {
    const { customerId, email } = event.payload;

    // Validate customer exists
    const customer = customerAggregate.state.get(state =>

```

```
const customer = customerAggregate.state.get(state =>
  (state.customers || {})[customerId]
);

if (!customer) {
  return {
    success: false,
    reason: "customer-not-found"
  };
}

// Update email
customerAggregate.state.set(state => {
  const customers = state.customers || {};

  return {
    ...state,
    customers: {
      ...customers,
      [customerId]: {
        ...customers[customerId],
        email,
        emailUpdatedAt: Date.now()
      }
    }
  };
});

// Broadcast email changed event
customerAggregate.broadcast({
  type: "customer-email-changed",
  payload: {
    customerId,
    previousEmail: customer.email,
    newEmail: email
  }
});

return {
  success: true,
  customerId
};
});
```

This pattern demonstrates how Happen can combine reactive event handling with consistent state management in domain-driven designs.

# Open Source

# Contributing

Don't be shy!

## Contributing to Happen

Thank you for your interest in contributing to Happen! As a framework built on the philosophy of radical simplicity, we value contributions that maintain this principle while extending the framework's capabilities.

## Core Philosophy

Before contributing, please familiarize yourself with Happen's core philosophy:

- **Radical Simplicity:** True power emerges from simplicity rather than complexity
- **Pure Causality:** Events form natural causal chains that define system behavior
- **Decentralized Intelligence:** Smart systems emerge from simple nodes making local decisions
- **Composable Primitives:** Complex behaviors emerge from simple, understandable parts
- **Runtime Transparency:** Direct access to the underlying runtime environments

Our goal is to maintain a framework with minimal primitives that compose to create powerful capabilities. We believe the most elegant solutions often emerge not from adding complexity, but from discovering the right minimal abstractions.

## How to Contribute

## HOW TO CONTRIBUTE

### Reporting Issues

When reporting issues, please include:

1. A clear description of the problem
2. Steps to reproduce the issue
3. Expected vs. actual behavior
4. Version information (Happen version, runtime environment, OS)
5. Minimal code example that demonstrates the issue

### Feature Requests

We welcome feature requests that align with Happen's philosophy. When suggesting features:

1. Describe the problem you're trying to solve before proposing a solution
2. Explain how the feature aligns with Happen's philosophy of radical simplicity
3. Consider whether the feature could be implemented as a composable extension rather than a core addition
4. If possible, include examples of how the feature would be used

### Pull Requests

When submitting pull requests:

1. Create a branch with a descriptive name (e.g., `feature/event-replay` or `fix/node-discovery`)
2. Include tests for new functionality
3. Ensure all tests pass

4. Update documentation to reflect changes
5. Keep PRs focused on a single concern
6. Follow the existing code style and patterns

## Development Setup

1. Fork and clone the repository

```
git clone https://github.com/RobAntunes/happen.git  
cd happen
```

2. Install dependencies

```
npm install
```

3. Run tests

```
npm test
```

## Coding Guidelines

### Simplicity First

- Before adding code, ask: "Is this the simplest possible solution?"
- Prefer fewer, well-designed primitives over many specialized ones
- Follow the principle of "Do One Thing Well"

### Event-Driven Design

- Maintain the event-driven nature of the framework



- Ensure causality is preserved in all operations
- Keep nodes focused on single responsibilities

## API Design

- APIs should be intuitive and consistent with existing patterns
- Prefer composable functions over complex objects
- Maintain backward compatibility when possible

## Testing

- Write unit tests for all new functionality
- Include integration tests for complex interactions
- Test across different runtime environments when applicable

## Documentation

Documentation is crucial for a framework like Happen. When adding or changing features:

1. Update the relevant documentation files
2. Include clear, concise examples
3. Explain not just how to use a feature, but why and when to use it
4. Ensure code examples are tested and working

## Review Process

All contributions go through a review process:

1. Automated checks (linting, tests)

2. Code review by maintainers
3. Possible revisions based on feedback
4. Final approval and merge

## Community Guidelines

We strive to maintain a welcoming and inclusive community:

- Be respectful and considerate in all interactions
- Focus on the ideas being discussed, not the person presenting them
- Assume good intentions from other contributors
- Help others learn and grow through constructive feedback

## Recognition

All contributors are valued and will be recognized in our documentation and release notes. Significant contributors may be invited to join as maintainers.

## Questions?

If you have questions about contributing, please [open an issue](#) with the label "question".

Thank you for helping make Happen better while embracing the philosophy of simplicity!