HyperFlow Client SDK Guide

HyperFlow Client SDK Guide

Overview

The HyperFlow Client SDK is a JavaScript/TypeScript library that simplifies building custom applications powered by HyperFlow flow-graphs. It handles all the complexity of the Control API communication, state management, streaming, and interaction lifecycle, letting you focus on your application's UI and business logic.

What It Provides:

- **Core SDK**: Headless client for any JavaScript environment (browser, Node.js, React Native)
- **React Integration**: Hooks and state management for React applications
- **Type Safety**: Full TypeScript support with complete type definitions
- **Production-Tested**: Based on HyperFlow's own embeddable chatbot implementation

Why Use the SDK?

Without SDK (Direct Control API):

- 800+ lines of integration code
- Manual management of interaction specs, step indices, streaming loops
- Easy to get wrong (missing paramSpec, tight polling, etc.)
- See: Building Custom Apps for the manual approach

With SDK:

- 50 lines of code
- All Control API complexity handled automatically
- Correct patterns guaranteed (long-polling, streaming, state management)
- Just configure and use

```
// Without SDK: 800+ lines of complex polling, state management, spec ha
ndling...

// With SDK:
const { messages, sendMessage, isReady } = useHyperFlow({
  apiKey: 'hf_...',
  flowGraphID: '67ca914369e9b8a95c967fdf',
});

// Done! Everything just works.
```

Installation

Current Status

The SDK is currently integrated into the HyperFlow monorepo. To use it in your application:

Option 1: Direct Import (if building within HyperFlow monorepo)

```
import { HyperFlowClient } from '@mirinae/sdk/client';
import { useHyperFlow } from '@mirinae/sdk/react';
```

Option 2: Copy SDK Files (for external projects)
Copy the `src/sdk/` directory to your project and adjust imports.

Future: Standalone NPM package `@hyperflow/client-sdk` (planned)

Quick Start (React)

1. Get Your API Key

1. Log into HyperFlow: https://hyperflow-ai.com
2. Navigate to **Account Settings → API Configuration**
3. Click **Generate New API Key**
4. Configure **Allowed Origins** for client-side usage:

```
https://myapp.com
https://www.myapp.com
http://localhost:3000
```

5. Copy your API key (starts with `hf_`)

See: API Key Guide for origin restriction details

2. Get Your FlowGraph ID

1. Create and publish a flow-graph in HyperFlow IDE
2. Copy the flowGraphID from the publish manager (24-character hex string)
3. Or use the flowGraphID from the URL: `/hyperflow/editor?fg=<flowGraphID>`

3. Basic React Chat Component

```
import React, { useState } from 'react';
import { useHyperFlow } from '@mirinae/sdk/react';

function MyChat() {
    const [inputValue, setInputValue] = useState('');

    // Initialize HyperFlow
    const {
        messages,
        sendMessage,
        isReady,
        isGenerating,
        promptInteractionSpec,
        error
    } = useHyperFlow({
        baseURL: 'https://hyperflow-ai.com',
        apiKey: 'hf_your_api_key_here',
        flowGraphID: '67ca914369e9b8a95c967fdf',
        queryParams: {
            userId: 'user-123',
            language: 'en'
        },
```

```
    debug: true,  // Enable during development for verbose logging

    // Optional: Callbacks for custom handling
    onGeneratedText: (text, metadata) => {
      console.log('Tokens used:', metadata?.usage?.totalTokens);
    }
  });

  const handleSend = async () => {
    if (!inputValue.trim() || !isReady) return;

    try {
      await sendMessage(inputValue);
      setInputValue('');
    } catch (err) {
      console.error('Failed to send:', err);
    }
  };

  return (
    <div>
      {/* Message List */}
      <div>
        {messages.map((msg, i) => (
          <div key={i}>
            <strong>{msg.type === 'user' ? 'You' : 'Bot'}:</strong>
            {msg.text}
          </div>
        ))}
      </div>

      {/* Error Display */}
      {error && <div style={{ color: 'red' }}>Error: {error.message}</div>}

      {/* Input */}
      <input
        type="text"
        value={inputValue}
```

```
            onChange={e => setInputValue(e.target.value)}
            onKeyPress={e => e.key === 'Enter' && handleSend()}
            placeholder={promptInteractionSpec?.placeholder || 'Type a mes
sage...'}
            disabled={!isReady || isGenerating}
          />
          <button onClick={handleSend} disabled={!isReady || isGenerating}>
            {isGenerating ? 'Sending...' : 'Send'}
          </button>
        </div>
    );
}
```

That's it! The SDK handles:

- ✅ Session initialization
- ✅ Long-polling loop
- ✅ Interaction spec management
- ✅ Streaming LLM responses
- ✅ Step index tracking
- ✅ Error handling

Core SDK (Headless)

For non-React applications or when you need full control, use the headless `HyperFlowClient`.

Basic Usage

```
import { HyperFlowClient } from '@mirinae/sdk/client';

// Initialize client
const client = new HyperFlowClient({
    baseURL: 'https://hyperflow-ai.com',
    apiKey: 'hf_your_api_key',
    flowGraphID: '67ca914369e9b8a95c967fdf',
    queryParams: {
        userId: 'user-123',
```

```
        language: 'en'
    }
});

// Set up event handlers
client.on('start', (flowGraphName) => {
    console.log(`Started: ${flowGraphName}`);
});

client.on('message', (text) => {
    console.log(`Bot: ${text}`);
});

client.on('generatedText', (text, metadata, sessionStepID) => {
    console.log(`LLM: ${text}`);
    console.log(`Tokens: ${metadata?.usage?.totalTokens}`);
    updateUI(text);
});

client.on('interaction', (spec) => {
    if (spec.type === 'prompt') {
        enableUserInput(spec.placeholder);
    }
});

client.on('error', (error) => {
    console.error('SDK Error:', error);
});

// Start session
await client.start();

// Send user message (when ready)
await client.sendMessage('Hello!');

// Send feedback
await client.sendFeedback('positive', sessionStepID);
```

```
// End session
await client.end();
```

Event Reference

Event	Arguments	Description
`start`	`(flowGraphName: string)`	Session initialized
`message`	`(text: string)`	Flow-graph sent a message
`busyMessage`	`(text: string)`	Status update during processing
`generation.start`	`(sessionStepID: string, busyMessage?: string)`	LLM generation starting
`streamingUpdate`	`(text: string)`	Incremental LLM output (for typing animation)
`generatedText`	`(text: string, metadata?: Metadata, sessionStepID?: string)`	Complete LLM response
`interaction`	`(spec: InteractionSpec)`	Flow-graph needs user input
`media`	`(media: MediaContent)`	Flow-graph sent media
`references`	`(refs: Reference[])`	RAG sources
`host.postMessage`	`(message: any, targetOrigin: string)`	Data for host app (e.g., chat titles)
`exception`	`(exception: any)`	Error occurred
`end`	`()`	Session ended
`error`	`(error: Error)`	SDK error
`ready`	`()`	Ready to receive user input

Methods Reference

```
class HyperFlowClient {
    // Session management
    async start(options?: { resumeFromSessionID?: string }): Promise<void>
    async end(): Promise<void>

    // User interactions
    async sendMessage(text: string): Promise<void>
    async uploadFile(file: File, uploads: any[], fileName: string): Promise<voi
```

```
d>
    async selectBranch(edgeIndex: number): Promise<void>

    // Feedback
    async sendFeedback(rating: 'positive' | 'negative', sessionStepID?: strin
g): Promise<void>

    // Utilities
    updateQueryParams(newParams: Record<string, any>): void
    getFileURL(fileId: string): string  // Generate URL for uploaded file

    // State getters
    get sessionID(): string | null
    get isReady(): boolean  // Has prompt interaction spec available
    get chatHistory(): ChatMessage[]
    get interactionSpecs(): { prompt, fileUpload, branchChoice }

    // Event emitter
    on(event: string, handler: Function): void
    off(event: string, handler: Function): void
}
```

Lifecycle Behavior:

- `start()` - Automatically ends existing session before starting new one (unless resuming)
- `start({ resumeFromSessionID })` - Uses `/control/refresh` endpoint to continue existing session (preserves sessionID)
- `end()` - Safe to call anytime (no-op if no session exists)
- Both methods support debug logging when `debug: true`

Session Continuity:
When you call `start({ resumeFromSessionID: 'abc123' })`, the SDK automatically uses the `/control/refresh` endpoint instead of `/control/start`. This preserves the original sessionID and continues the session from where it left off, rather than creating a new session with copied history.

React Integration

useHyperFlow Hook

The main React hook provides a clean, simple API:

```
import { useHyperFlow } from '@mirinae/sdk/react';

const {
  // State
  messages,          // Chat history
  isGenerating,      // Is LLM currently generating?
  isReady,           // Can user send message? (has prompt spec)
  sessionID,         // Current session ID
  busyMessage,       // Current busy/status message
  error,             // Last error (if any)

  // Interaction specs
  promptInteractionSpec,
  fileUploadInteractionSpec,
  branchChoiceInteractionSpec,

  // Actions
  sendMessage,       // (text: string) => Promise<void>
  uploadFile,        // (file, uploads, fileName) => Promise<void>
  selectBranch,      // (edgeIndex: number) => Promise<void>
  sendFeedback,      // (rating, sessionStepID?) => Promise<void>

  // Utilities
  start,             // (options?) => Promise<void> - Start new session
  updateQueryParams, // (params: object) => void - Update query params
mid-session
  reset,             // () => void - Reset to initial state
  clearHistory,      // () => void - Clear chat history
  end,               // () => Promise<void> - End session
} = useHyperFlow({
  baseURL: 'https://hyperflow-ai.com',
  apiKey: 'hf_...',
  flowGraphID: '67ca914369e9b8a95c967fdf',
  queryParams: { userId: 'user-123' },
```

```
    // Optional configuration
    resumeFromSessionID: 'previous-session-id',
    autoStart: true,  // Default: true

    // Optional lifecycle callbacks
    onMessage: (text) => { /* ... */ },
    onGeneratedText: (text, metadata) => { /* ... */ },
    onInteraction: (spec) => { /* ... */ },
    onTitleUpdate: (title) => { /* ... */ },
    onError: (error) => { /* ... */ },
    onMedia: (media) => { /* ... */ },
    onReferences: (refs) => { /* ... */ },
});
```

Complete React Example

```
import React, { useState, useEffect } from 'react';
import { useHyperFlow } from '@mirinae/sdk/react';

interface ChatProps {
    flowGraphID: string;
    apiKey: string;
    userId: string;
}

export const HyperFlowChat: React.FC<ChatProps> = ({ flowGraphID, apiK
ey, userId }) => {
    const [inputValue, setInputValue] = useState('');
    const [conversationId, setConversationId] = useState<string | null>(null);

    const {
        messages,
        sendMessage,
        isReady,
        isGenerating,
        sessionID,
        error,
        promptInteractionSpec,
```

```
} = useHyperFlow({
  baseURL: 'https://hyperflow-ai.com',
  apiKey,
  flowGraphID,
  queryParams: { userId, language: 'en' },

  // Save messages to your database
  onGeneratedText: async (text, metadata) => {
    if (conversationId) {
      await fetch('/api/messages/save', {
        method: 'POST',
        headers: { 'Content-Type': 'application/json' },
        body: JSON.stringify({
          conversationId,
          role: 'assistant',
          content: text,
          tokens: metadata?.usage?.totalTokens,
          model: metadata?.model?.name,
        }),
      });
    }
  },

  // Auto-generated chat titles
  onTitleUpdate: async (title) => {
    if (conversationId) {
      await fetch(`/api/conversations/${conversationId}`, {
        method: 'PATCH',
        headers: { 'Content-Type': 'application/json' },
        body: JSON.stringify({ title }),
      });
    }
  },

  onError: (error) => {
    console.error('HyperFlow error:', error);
  },
});
```

```
// Create conversation in your DB when session starts
useEffect(() ⇒ {
  if (sessionID && !conversationId) {
    fetch('/api/conversations/create', {
      method: 'POST',
      headers: { 'Content-Type': 'application/json' },
      body: JSON.stringify({
        userId,
        hyperflowSessionID: sessionID,
      }),
    })
      .then(res ⇒ res.json())
      .then(data ⇒ setConversationId(data.conversationId));
  }
}, [sessionID]);

const handleSend = async () ⇒ {
  if (!inputValue.trim() || !isReady || isGenerating) return;

  const messageText = inputValue;
  setInputValue('');

  try {
    // Save user message to your DB
    if (conversationId) {
      await fetch('/api/messages/save', {
        method: 'POST',
        headers: { 'Content-Type': 'application/json' },
        body: JSON.stringify({
          conversationId,
          role: 'user',
          content: messageText,
        }),
      });
    }

    // Send to HyperFlow
```

```
      await sendMessage(messageText);
    } catch (err) {
      console.error('Failed to send message:', err);
      setInputValue(messageText); // Restore input on error
    }
  };

  return (
    <div style={{ maxWidth: '800px', margin: '0 auto', padding: '20px' }}>
      {/* Header */}
      <div style={{ marginBottom: '20px' }}>
        <h2>HyperFlow Chat</h2>
        <p style={{ fontSize: '14px', color: '#666' }}>
          {error ? (
            <span style={{ color: 'red' }}>Error: {error.message}</span>
          ) : isGenerating ? (
            'Generating...'
          ) : isReady ? (
            <span style={{ color: 'green' }}>Ready</span>
          ) : (
            'Connecting...'
          )}
        </p>
      </div>

      {/* Messages */}
      <div style={{
        height: '500px',
        overflowY: 'auto',
        border: '1px solid #ddd',
        borderRadius: '8px',
        padding: '16px',
        marginBottom: '16px'
      }}>
        {messages.map((msg, i) => (
          <div
            key={i}
            style={{
```

```
                marginBottom: '12px',
                textAlign: msg.type === 'user' ? 'right' : 'left',
              }}
            >
              <div
                style={{
                  display: 'inline-block',
                  maxWidth: '70%',
                  padding: '8px 12px',
                  borderRadius: '12px',
                  backgroundColor: msg.type === 'user' ? '#007bff' : '#f0
f0f0',
                  color: msg.type === 'user' ? 'white' : 'black',
                }}
              >
                {msg.text}
              </div>
            </div>
          ))}
        </div>

        {/* Input */}
        <div style={{ display: 'flex', gap: '8px' }}>
          <input
            type="text"
            value={inputValue}
            onChange={e => setInputValue(e.target.value)}
            onKeyPress={e => e.key === 'Enter' && !e.shiftKey && handleSe
nd()}
            placeholder={promptInteractionSpec?.placeholder || 'Type a m
essage...'}
            disabled={!isReady || isGenerating}
            style={{
              flex: 1,
              padding: '10px',
              fontSize: '14px',
              border: '1px solid #ddd',
              borderRadius: '4px',
```

```
            }}
          />
          <button
            onClick={handleSend}
            disabled={!isReady || isGenerating || !inputValue.trim()}
            style={{
              padding: '10px 20px',
              fontSize: '14px',
              backgroundColor: '#007bff',
              color: 'white',
              border: 'none',
              borderRadius: '4px',
              cursor: isReady && !isGenerating ? 'pointer' : 'not-allowed',
            }}
          >
            Send
          </button>
        </div>
      </div>
    );
}
```

Security: Client-Side API Keys

Origin Restrictions

For safe client-side usage, configure origin restrictions on your API key:

In HyperFlow Account Settings → API Configuration:

```
Allowed Origins:
https://myapp.com
https://www.myapp.com
http://localhost:3000
```

How It Works:

- Browser sends `Origin` header with every request
- HyperFlow validates origin against your allowed list

- Even if someone copies your API key, it only works from your domains

Best Practices:

- ✅ Use HTTPS in production (not HTTP)
- ✅ List all subdomains explicitly (no wildcards)
- ✅ Include localhost for development
- ✅ Separate API keys for dev vs. production
- ❌ Don't leave allowed origins empty for client-side keys

See: Origin Restrictions Release Notes for complete details

Example: Environment-Based Configuration

```
// .env.local (development)
REACT_APP_HYPERFLOW_API_KEY=hf_dev_key_with_localhost
REACT_APP_HYPERFLOW_FLOWGRAPH_ID=67ca914369e9b8a95c967fdf
REACT_APP_HYPERFLOW_BASE_URL=http://localhost:2000

// .env.production
REACT_APP_HYPERFLOW_API_KEY=hf_prod_key_with_myapp_domain
REACT_APP_HYPERFLOW_FLOWGRAPH_ID=67ca914369e9b8a95c967fdf
REACT_APP_HYPERFLOW_BASE_URL=https://hyperflow-ai.com
```

```
// App.tsx
const config = {
    apiKey: process.env.REACT_APP_HYPERFLOW_API_KEY,
    flowGraphID: process.env.REACT_APP_HYPERFLOW_FLOWGRAPH_ID,
    baseURL: process.env.REACT_APP_HYPERFLOW_BASE_URL,
};

// Use in component
const chat = useHyperFlow(config);
```

Configuration Options

HyperFlowClientConfig

```
interface HyperFlowClientConfig {
  baseURL: string;            // HyperFlow API base URL
  apiKey: string;             // Your API key (from Account Settings)
  flowGraphID: string;        // Published flow-graph ID (24-char hex)
  queryParams?: Record<string, any>;  // Optional context data
  language?: string;          // Default: 'en' (for i18n)
  debug?: boolean;            // Enable debug logging (default: false)
  initialMessages?: ChatMessage[];  // Seed messages array (for UI restora
tion)
}
```

Initial Messages:

Populate the SDK's messages array on initialization (for restoring UI from persistence):

```
const savedMessages = JSON.parse(localStorage.getItem('chat') || '[]');

useHyperFlow({
  apiKey: 'hf_...',
  flowGraphID: '...',
  initialMessages: savedMessages,  // ← Seed messages array
});

// SDK messages array now includes savedMessages + new messages
```

Use with `resumeFromSessionID` for complete restoration:

```
useHyperFlow({
  resumeFromSessionID: savedSessionID,    // Restore LLM context
  initialMessages: savedMessages,         // Restore UI messages
});
```

Debug Mode:

Enable verbose logging of all Control API requests and responses:

```
useHyperFlow({
  apiKey: 'hf_...',
```

```
    flowGraphID: '...',
    debug: true,  // ← Logs all API calls to console
});
```

Console output:

```
[HyperFlow SDK] POST /api/flowgraph/control/start
Request: {
  "flowGraphID": "67ca914369e9b8a95c967fdf",
  "queryParams": { "userId": "user-123" }
}
Response: {
  "success": true,
  "data": {
    "sessionID": "68ecb5d0c96c227ef841b63d",
    "stepIndex": 0,
    "responses": [...]
  }
}
```

Recommendation: Enable during development, disable in production.

Query Parameters

Pass context to your flow-graph via `queryParams`:

```
useHyperFlow({
    // ...
    queryParams: {
        userId: 'user-123',          // User identification
        language: 'en',              // Language preference
        sessionType: 'support',      // Custom context
        metadata: {                  // Any JSON-serializable data
            source: 'mobile-app',
            version: '1.2.3'
        }
    }
});
```

Access in Flow-Graph:
Use the "Get Query Param" node to read these values and customize behavior.

Persistence and Updates:
Query params are saved in the session and persist across requests. You can update them mid-session:

```
const { updateQueryParams } = useHyperFlow({
    queryParams: { userId: 'user-123', language: 'en' },
});

// Update mid-session (incremental merge)
updateQueryParams({
    conversationId: 'conv-456',
    language: 'es'  // Override
});

// Next progress call includes:
// { userId: 'user-123', conversationId: 'conv-456', language: 'es' }
```

Merge Behavior:

- Parameters are merged client-side before sending
- Server also merges with session.queryParams
- New values override existing ones
- Use to update context dynamically (conversation switching, preferences, etc.)

Resume from Previous Session

```
useHyperFlow({
    // ...
    resumeFromSessionID: 'previous-session-id',
});
```

HyperFlow will copy the chat history from the previous session, maintaining conversation context.

Advanced Features

File Uploads

```
const { uploadFile, fileUploadInteractionSpec, getFileURL } = useHyperFlo
w({...});

// When file upload interaction is available
if (fileUploadInteractionSpec) {
  const handleFileUpload = async (e: React.ChangeEvent<HTMLInputElem
ent>) => {
    const file = e.target.files?.[0];
    if (!file) return;

    // Upload file to HyperFlow
    const formData = new FormData();
    formData.append('org_id', fileUploadInteractionSpec.orgID);
    formData.append('file', file);

    const response = await fetch('https://hyperflow-ai.com/api/hyperflow/
uploadContent', {
      method: 'POST',
      body: formData,
    });

    const { data } = await response.json();
    const { uploads, filename } = data;

    // Send upload reference to flow-graph
    await uploadFile(file, uploads, filename);
  };

  return (
    <input
      type="file"
      onChange={handleFileUpload}
      accept={fileUploadInteractionSpec.fileType || '*/*'}
    />
```

```
    );
}
```

Accessing Uploaded Files

The SDK provides the `getFileURL()` helper to generate URLs for uploaded files:

```
const { getFileURL, messages } = useHyperFlow({...});

// Display uploaded files from chat history
{messages.map((msg, i) => (
  msg.type === 'user' && msg.uploads && (
    <div key={i}>
      <p>{msg.text}</p>
      {msg.uploads.map((upload, j) => {
        const fileURL = upload.url || getFileURL(upload.data);

        return upload.mimetype?.startsWith('image/') ? (
          <img key={j} src={fileURL} alt={upload.fileName} />
        ) : (
          <a key={j} href={fileURL} download={upload.fileName}>
            Download {upload.fileName}
          </a>
        );
      })}
    </div>
  )
))}
```

The SDK automatically adds URLs to uploads in chat history. See File Upload Handling Guide for complete examples.

Branch Selection

```
const { selectBranch, branchChoiceInteractionSpec } = useHyperFlow
({...});

if (branchChoiceInteractionSpec) {
  return (
```

```
      <div>
        <p>Choose an option:</p>
        {branchChoiceInteractionSpec.branches.map((branch, i) ⇒ (
          <button key={i} onClick={() ⇒ selectBranch(i)}>
            {branch.label}
          </button>
        ))}
      </div>
    );
}
```

User Feedback

```
const { sendFeedback } = useHyperFlow({...});

// Render feedback buttons after LLM responses
{messages.map((msg, i) ⇒ (
  msg.type === 'generator' && (
    <div>
      <button onClick={() ⇒ sendFeedback('positive', msg.sessionStepI
D)}>
        👍 Helpful
      </button>
      <button onClick={() ⇒ sendFeedback('negative', msg.sessionStepI
D)}>
        👎 Not Helpful
      </button>
    </div>
  )
))}
```

Streaming with Typing Animation

The SDK automatically handles streaming and provides smooth typing animation:

```
const { messages, isGenerating } = useHyperFlow({...});
```

```
// Message types:
// - 'streamedGenerator': Currently streaming (text updates in real-time)
// - 'generator': Final complete response

{messages.map((msg, i) => (
  <div key={i}>
    {msg.type === 'streamedGenerator' && (
      <div>
        {msg.text}
        <span className="typing-cursor">|</span>
      </div>
    )}
    {msg.type === 'generator' && (
      <div>{msg.text}</div>
    )}
  </div>
))}
```

The SDK uses adaptive typing speed:

- Fast when far behind (catches up quickly)
- Slow when close to real-time (smooth animation)
- ~60fps update rate for smooth visuals

Metadata and Token Usage

```
const { onGeneratedText } = useHyperFlow({
  // ...
  onGeneratedText: (text, metadata) => {
    // Track tokens for billing
    const tokens = metadata?.usage?.totalTokens || 0;
    console.log(`Used ${tokens} tokens`);

    // Track model used
    const model = metadata?.model?.name || 'unknown';
    console.log(`Model: ${model}`);

    // Performance tracking
```

```
      const elapsedMs = (metadata?.timing?.elapsedNs || 0) / 1_000_000;
      console.log(`Response time: ${elapsedMs.toFixed(0)}ms`);

      // Save to analytics
      trackLLMUsage({
        conversationId,
        tokens,
        model,
        elapsed: elapsedMs,
      });
    },
});
```

Host Messages (Chat Titles)

```
const { onTitleUpdate } = useHyperFlow({
    // ...
    onTitleUpdate: (title) => {
      // Flow-graph auto-generated a chat title
      console.log('New title:', title);

      // Update your UI
      setConversationTitle(title);

      // Save to database
      saveTitle(conversationId, title);
    },
});
```

Flow-graphs can automatically generate chat titles based on conversation content using the "Message to Host" node.

Dynamic Query Parameters

Update query parameters mid-session to change context:

```
const { updateQueryParams } = useHyperFlow({
    queryParams: { userId: 'user-123', language: 'en' },
});
```

```
// Switch to different conversation
const handleSwitchConversation = (newConvId: string) => {
    updateQueryParams({
        conversationId: newConvId,
        // Merged with existing params
    });

    // Next message will include updated params
};

// Update user preferences
const handleLanguageChange = (lang: string) => {
    updateQueryParams({ language: lang });
};

// Add temporary flags
const handleEnableDebugMode = () => {
    updateQueryParams({ debugMode: true });
};
```

Use Cases:

- Switch between conversations without new session
- Update user preferences (language, theme)
- Pass dynamic context to flow-graph
- A/B testing variants
- Feature flags

Session Restore Patterns

⚠️ Critical Understanding: Two Separate Concerns

When resuming a conversation, you need to handle **TWO different things**:

Concern	What It Is	Handled By
UI Display	Chat messages the user sees	**Your app** (must persist)

Concern	What It Is	Handled By
LLM Context	Conversation history for AI responses	**HyperFlow** (via resumeFromSessionID)

What `resumeFromSessionID` Does (and Doesn't Do)

✅ **resumeFromSessionID DOES:**

- Restore server-side conversation history for LLM context
- Allow flow-graph to "remember" previous messages when generating responses
- Maintain conversation continuity for AI

❌ **resumeFromSessionID does NOT:**

- Restore UI message bubbles
- Populate the SDK's `messages` array
- Show chat history in your interface

The SDK's `messages` array only exists in memory during an active session. When you unmount or refresh, it's gone.

Message Persistence Requirements

To restore chat UI, you must persist messages in the **exact SDK format**:

```
interface ChatMessage {
    type: 'user' | 'assistant' | 'generator' | 'streamedGenerator' | 'message' |
'error' | 'media' | 'references';
    text?: string;
    sessionStepID?: string;
    metadata?: {
        usage?: { promptTokens: number; completionTokens: number; totalTok
ens: number };
        model?: { name: string; provider: string };
        timing?: { elapsedNs: number };
    };
    timestamp?: number;
    media?: { mimetype: string; data: string };
```

```
    references?: Reference[];
}
```

Pattern 1: Database Persistence with Initial Messages (Recommended)

Best for: Production applications with backend

```
import { useState, useEffect } from 'react';
import { useHyperFlow } from '@mirinae/sdk/react';

function Chat({ conversationId }) {
    const [conversation, setConversation] = useState(null);
    const [savedMessages, setSavedMessages] = useState([]);

    // Load conversation and messages from database
    useEffect(() => {
        Promise.all([
            fetch(`/api/conversations/${conversationId}`).then(r => r.json()),
            fetch(`/api/messages?conversationId=${conversationId}`).then(r =>
r.json()),
        ]).then(([conv, msgs]) => {
            setConversation(conv);
            setSavedMessages(msgs);
        });
    }, [conversationId]);

    // Initialize SDK with saved messages
    const { messages, sendMessage, sessionID } = useHyperFlow({
        apiKey: 'hf_...',
        flowGraphID: '...',
        resumeFromSessionID: conversation?.hyperflowSessionID,  // LLM con
text
        initialMessages: savedMessages,  // UI display - populate messages ar
ray

        // Save sessionID when created
        onStart: async () => {
```

```
        if (sessionID) {
          await fetch(`/api/conversations/${conversationId}`, {
            method: 'PATCH',
            body: JSON.stringify({ hyperflowSessionID: sessionID }),
          });
        }
      },
    });

    // Save messages after each update
    useEffect(() => {
      if (messages.length > 0) {
        // Persist SDK messages array to your database
        fetch('/api/messages/bulk', {
          method: 'POST',
          body: JSON.stringify({
            conversationId,
            messages,
          }),
        });
      }
    }, [messages]);

    const handleSend = async (text: string) => {
      await sendMessage(text);
      // Messages auto-saved via useEffect above
    };

    return (
      <div>
        {/* Single source of truth - SDK messages array */}
        {messages.map((msg, i) => (
          <div key={i}>{msg.text}</div>
        ))}
      </div>
    );
}
```

Key Points:

- ✅ Single source of truth (SDK messages array)
- ✅ Messages restored on mount via `initialMessages`
- ✅ Auto-saved via useEffect (standard React pattern)
- ✅ resumeFromSessionID provides LLM context
- ✅ Works across devices

Cons:

- ❌ Requires backend database

Pattern 2: localStorage with Initial Messages

Best for: Simple apps without backend, single-device usage

```
import { useEffect } from 'react';
import { useHyperFlow } from '@mirinae/sdk/react';

function Chat({ conversationId }) {
  const storageKey = `chat-${conversationId}`;

  // Load saved messages and sessionID
  const savedMessages = JSON.parse(localStorage.getItem(storageKey) |
| '[]');
  const savedSessionID = localStorage.getItem(`session-${conversationI
d}`);

  const { messages, sendMessage, sessionID, reset, start } = useHyperFlo
w({
    apiKey: 'hf_...',
    flowGraphID: '...',
    resumeFromSessionID: savedSessionID,    // LLM context
    initialMessages: savedMessages,         // UI display
  });

  // Auto-save messages after each update
  useEffect(() => {
    localStorage.setItem(storageKey, JSON.stringify(messages));
```

```
        if (sessionID) {
            localStorage.setItem(`session-${conversationId}`, sessionID);
        }
    }, [messages, sessionID]);

    const handleNewChat = async () => {
        localStorage.removeItem(storageKey);
        localStorage.removeItem(`session-${conversationId}`);
        reset();
        await start();
    };

    return (
        <div>
            <button onClick={handleNewChat}>New Chat</button>
            {/* Single source of truth - SDK messages array */}
            {messages.map((msg, i) => (
                <div key={i}>{msg.text}</div>
            ))}
        </div>
    );
}
```

Key Points:

- ✅ Single source of truth (SDK messages)
- ✅ Simple localStorage persistence
- ✅ Restored via `initialMessages`
- ✅ Standard React patterns (useEffect)

Pros:

- ✅ No backend required
- ✅ Clean, simple code
- ✅ Full UI restoration

Cons:

- ❌ localStorage only (no cross-device)

- ❌ Storage size limits (~5MB)

Pattern 3: Component Remount (Simplest - No Persistence)

Best for: Apps that don't need conversation history restoration

Don't use `resumeFromSessionID` at all—just start fresh each time:

```
function ChatApp() {
  const [chatKey, setChatKey] = useState(0);

  const { messages, sendMessage } = useHyperFlow({
    apiKey: 'hf_...',
    flowGraphID: '...',
    // No resumeFromSessionID - always fresh
  });

  const handleNewChat = () => {
    setChatKey(k => k + 1);  // Remount = fresh session
  };

  return (
    <div>
      <button onClick={handleNewChat}>New Chat</button>
      <Chat key={chatKey} />  {/* Remounts on key change */}
    </div>
  );
}
```

Pros:

- ✅ Simple - no persistence code
- ✅ Always clean state
- ✅ Good for demos/prototypes

Cons:

- ❌ Conversation lost on refresh
- ❌ No cross-device continuity
- ❌ Not suitable for production

How Message Restoration Works

Using `initialMessages` (v1.0+):

The SDK now supports populating the messages array on initialization:

```
const savedMessages = loadFromStorage();  // Your choice: localStorage, DB, etc.

const { messages } = useHyperFlow({
    initialMessages: savedMessages,  // ← Populates messages array
});

// messages array includes savedMessages + new messages
```

Persistence is still your responsibility:

- SDK provides `initialMessages` to seed the array
- You choose storage (localStorage, IndexedDB, database, etc.)
- You save via useEffect (standard React pattern)
- You load and pass via `initialMessages`

Why this approach:

- ✅ SDK stays lightweight (no built-in persistence)
- ✅ Developer controls storage mechanism
- ✅ Works with any backend (SQL, NoSQL, localStorage, etc.)
- ✅ Single source of truth (SDK messages array)
- ✅ Standard React patterns

Pattern Comparison

Feature	DB Persistence	localStorage	No Persistence
UI Restoration	✅ Full	✅ Full	❌ Lost on refresh
LLM Context	✅ Via resume	✅ Via resume	❌ No history
Cross-device	✅ Yes	❌ No	❌ No
Storage	Your DB	localStorage	None
Complexity	Medium	Medium	Low

Feature	DB Persistence	localStorage	No Persistence
Best for	Production	Single-device apps	Demos/prototypes

Recommendation:

- **Production**: Pattern 1 (database persistence)
- **Prototypes/Demos**: Pattern 3 (no persistence)
- **Simple apps**: Pattern 2 (localStorage)

Starting a New Chat

When users want to start a fresh conversation, you have several options depending on your use case.

Option 1: Clear Messages Only

Best for: Clearing the display while keeping the session active

```
const { clearHistory } = useHyperFlow({...});

<button onClick={clearHistory}>
  Clear Messages
</button>
```

What happens:

- ✅ Clears displayed messages from UI
- ✅ Session remains active (sessionID preserved)
- ✅ Server-side conversation history maintained

Use when: User wants to clear the screen but you want to keep the conversation context for analytics or other purposes.

Option 2: Reset and Restart (Recommended)

Best for: Complete fresh start with new session, no page reload

```
const { reset, start } = useHyperFlow({...});

const handleNewChat = async () => {
  reset();        // Clear all SDK state
```

```
    await start();    // Start fresh session (auto-ends old session if needed)
};

<button onClick={handleNewChat}>
    New Chat
</button>
```

What happens:

- ✅ Clears all SDK state (messages, sessionID, interaction specs)
- ✅ Ends existing session automatically (if any)
- ✅ Starts new session with no history
- ✅ No page reload or component remount needed

Use when: User explicitly wants a completely new conversation without losing page state.

Note: `start()` is smart—if a session already exists and you're not resuming, it automatically ends the old session first.

Option 3: Component Remount Pattern (Cleanest)

Best for: React applications wanting clean state reset without page reload

```
function ChatApp() {
    const [chatKey, setChatKey] = useState(0);

    const handleNewChat = () => {
        setChatKey(prev => prev + 1);  // Forces unmount/remount
    };

    return (
        <div>
            <button onClick={handleNewChat}>New Chat</button>
            <ChatComponent key={chatKey} />
        </div>
    );
}

function ChatComponent() {
```

```
    // This component fully unmounts and remounts
    const { messages, sendMessage } = useHyperFlow({...});

    // useEffect cleanup will call client.end() automatically
    return <div>{/* chat UI */}</div>;
}
```

What happens:

- ✅ Component unmounts (cleanup calls `end()`)
- ✅ Component remounts (new SDK instance, new session)
- ✅ No page reload needed
- ✅ Completely fresh state

Use when: You want the cleanest reset without full page reload.

Option 4: With localStorage Cleanup

Best for: Apps using localStorage for session persistence

```
const { reset } = useHyperFlow({...});

const handleNewChat = () => {
    const flowGraphID = '67ca914369e9b8a95c967fdf';

    // Clear all cached session data
    localStorage.removeItem(`hyperflow-session-${flowGraphID}`);
    localStorage.removeItem(`hyperflow-fg-${flowGraphID}`);
    localStorage.removeItem(`hyperflow-state-${flowGraphID}`);

    // Reset SDK state
    reset();

    // Navigate with ?new=true to prevent restore
    window.location.href = window.location.pathname + '?new=true';
};

<button onClick={handleNewChat}>New Chat</button>
```

What happens:

- ✅ Removes all localStorage entries
- ✅ Prevents automatic session restore
- ✅ URL param signals "don't restore"
- ✅ Forces completely fresh session

Use when: Using localStorage session restore (Pattern 2 or 3 above).

Method Comparison

Method	Clears UI	Ends Session	Reloads Page	Complexity
`clearHistory()`	✅	❌	❌	Low
`reset()` + `start()`	✅	✅	❌	Low
Component remount	✅	✅	❌	Medium
localStorage cleanup	✅	✅	✅	Medium

Recommended Approach

For most applications (simplest):

```
// Option 2: Reset and restart (no page reload or remount)
const { reset, start } = useHyperFlow({...});

const handleNewChat = async () => {
  reset();
  await start();  // Smart - auto-ends old session if needed
};

<button onClick={handleNewChat}>New Chat</button>
```

For multi-conversation UIs:

```
// Option 3: Component remount (preserves parent component state)
const [chatKey, setChatKey] = useState(0);

<button onClick={() => setChatKey(k => k + 1)}>
```

```
    New Chat
</button>
<Chat key={chatKey} />
```

Complete Example (Simple Pattern)

```
import { useState } from 'react';
import { useHyperFlow } from '@hyperflow/client-sdk/react';

function ChatApp() {
    const [conversationId, setConversationId] = useState<string | null>(null);

    const { messages, sendMessage, isReady, reset, start } = useHyperFlow
({
        apiKey: 'hf_...',
        flowGraphID: '...',
        queryParams: { conversationId },
    });

    const handleNewChat = async () => {
        // Create new conversation in your database
        const newConv = await fetch('/api/conversations/create', {
            method: 'POST',
        }).then(r => r.json());

        setConversationId(newConv.id);

        // Start fresh HyperFlow session
        reset();
        await start();
    };

    return (
        <div>
            <header>
                <h1>My Chat App</h1>
                <button onClick={handleNewChat}>+ New Chat</button>
            </header>
```

```
        <div>
          {messages.map((msg, i) => (
            <div key={i}>{msg.text}</div>
          ))}
          <input
            onKeyPress={e => {
              if (e.key === 'Enter') {
                sendMessage(e.currentTarget.value);
                e.currentTarget.value = '';
              }
            }}
            disabled={!isReady}
          />
        </div>
      </div>
    );
}
```

UI Component Patterns

The SDK focuses on Control API communication and doesn't include UI components. This keeps it lightweight and lets you customize the look and feel. Below are recommended implementations for common UI elements.

Markdown Rendering

For rendering LLM responses with rich formatting (like ChatGPT/Claude):

Install Dependencies:

```
npm install react-markdown remark-gfm remark-math rehype-katex
npm install katex  # For math equations
```

Component:

```
import React from 'react';
import ReactMarkdown from 'react-markdown';
import remarkGfm from 'remark-gfm';
import remarkMath from 'remark-math';
import rehypeKatex from 'rehype-katex';
```

```
import 'katex/dist/katex.min.css';

interface MarkdownRendererProps {
  content: string;
  className?: string;
}

export const MarkdownRenderer: React.FC<MarkdownRendererProps> =
({ content, className }) => {
  return (
    <div className={className}>
      <ReactMarkdown
        remarkPlugins={[remarkGfm, remarkMath]}
        rehypePlugins={[rehypeKatex]}
        components={{
          // Custom code block styling
          code({ node, inline, className, children, ...props }) {
            return inline ? (
              <code className="inline-code" {...props}>
                {children}
              </code>
            ) : (
              <pre className="code-block">
                <code className={className} {...props}>
                  {children}
                </code>
              </pre>
            );
          },
          // Open links in new tab
          a({ node, href, children, ...props }) {
            return (
              <a href={href} target="_blank" rel="noopener noreferrer"
{...props}>
                {children}
              </a>
            );
          },
```

```
            }}
          >
            {content}
          </ReactMarkdown>
        </div>
    );
};
```

CSS (optional):

```
.inline-code {
    background: #f0f0f0;
    padding: 2px 6px;
    border-radius: 3px;
    font-family: monospace;
    font-size: 0.9em;
}

.code-block {
    background: #f5f5f5;
    padding: 12px;
    border-radius: 6px;
    overflow-x: auto;
    margin: 8px 0;
}

.code-block code {
    font-family: 'Monaco', 'Courier New', monospace;
    font-size: 0.9em;
    line-height: 1.5;
}
```

Usage:

```
{messages.map((msg, i) => (
    msg.type === 'generator' && (
        <MarkdownRenderer content={msg.text} className="message-conte
nt" />
```

```
    )
))}
```

Features:

- ✅ GitHub-Flavored Markdown (tables, task lists, strikethrough)
- ✅ Math equations (LaTeX with KaTeX)
- ✅ Code blocks with syntax highlighting (add Prism.js for more)
- ✅ Links open in new tabs
- ✅ Sanitized HTML (safe from XSS)

Media Rendering

For displaying images, videos, and audio from flow-graph responses:

```
interface MediaRendererProps {
    media: {
        mimetype: string;
        data: string;  // URL or data URI
    };
    onLoad?: () => void;
}

export const MediaRenderer: React.FC<MediaRendererProps> = ({ media,
onLoad }) => {
    const { mimetype, data } = media;

    // Image (including SVG)
    if (mimetype.includes('image')) {
        if (mimetype.includes('svg')) {
            return (
                <div className="media-svg" dangerouslySetInnerHTML={{ __ht
ml: data }} />
            );
        }
        return (
            <div className="media-image">
                <img
```

```
            src={data}
            alt="Generated content"
            onLoad={onLoad}
            style={{
              maxWidth: '100%',
              height: 'auto',
              borderRadius: '8px',
              boxShadow: '0 2px 8px rgba(0,0,0,0.1)',
            }}
          />
        </div>
      );
    }

    // Video
    if (mimetype.includes('video')) {
      return (
        <div className="media-video">
          <video
            controls
            src={data}
            onLoadedData={onLoad}
            style={{
              maxWidth: '100%',
              height: 'auto',
              minHeight: '200px',
              borderRadius: '8px',
            }}
          />
        </div>
      );
    }

    // Audio
    if (mimetype.includes('audio')) {
      return (
        <div className="media-audio">
          <audio
```

```
                controls
                src={data}
                onLoadedData={onLoad}
                style={{
                  width: '100%',
                  height: '40px',
                }}
              />
            </div>
        );
    }

    return <div>Unsupported media type: {mimetype}</div>;
};
```

Usage:

```
{messages.map((msg, i) => (
    msg.type === 'media' && msg.media && (
        <MediaRenderer key={i} media={msg.media} onLoad={scrollToBotto
m} />
    )
))}
```

Loading Spinners

Option 1: Simple CSS Spinner (no dependencies)

```
export const Spinner: React.FC<{ size?: number }> = ({ size = 24 }) => {
    return (
        <div
            className="spinner"
            style={{
                width: size,
                height: size,
                border: '3px solid #f3f3f3',
                borderTop: '3px solid #007bff',
                borderRadius: '50%',
```

```
            animation: 'spin 1s linear infinite',
          }}
        />
    );
};

// Add to your global CSS:
// @keyframes spin {
//     0% { transform: rotate(0deg); }
//     100% { transform: rotate(360deg); }
// }
```

Option 2: Animated SVG

```
export const SpinningGear: React.FC = () => (
    <svg
        width="24"
        height="24"
        viewBox="0 0 24 24"
        xmlns="http://www.w3.org/2000/svg"
        style={{ animation: 'spin 2s linear infinite' }}
    >
        <path
            fill="currentColor"
            d="M12,1L9,4l3,3l3-3M4,9l-3,3l3,3l3-3M20,9l-3,3l3,3l3-3M12,15l-3,
3l3,3l3-3M12,9a3,3,0,1,1-3,3a3,3,0,0,1,3-3"
        />
    </svg>
);
```

Usage:

```
{isGenerating && (
    <div className="busy-indicator">
        <Spinner />
        <span>{busyMessage || 'Thinking...'}</span>
    </div>
)}
```

Copy-to-Clipboard Button

```
interface CopyButtonProps {
  text: string;
  onCopy?: () => void;
}

export const CopyButton: React.FC<CopyButtonProps> = ({ text, onCopy })
=> {
  const [copied, setCopied] = useState(false);

  const handleCopy = async () => {
    try {
      await navigator.clipboard.writeText(text);
      setCopied(true);
      setTimeout(() => setCopied(false), 2000);
      onCopy?.();
    } catch (err) {
      console.error('Failed to copy:', err);
    }
  };

  return (
    <button
      onClick={handleCopy}
      title={copied ? 'Copied!' : 'Copy to clipboard'}
      style={{
        padding: '4px 8px',
        border: '1px solid #e0e0e0',
        borderRadius: '4px',
        background: copied ? '#e8f5e9' : 'transparent',
        cursor: 'pointer',
        fontSize: '12px',
      }}
    >
      {copied ? '✓ Copied' : '📋 Copy'}
    </button>
```

```
  );
};
```

Usage:

```
{msg.type === 'generator' && (
  <div className="message-actions">
    <CopyButton text={msg.text} />
  </div>
)}
```

Feedback Buttons

```
interface FeedbackButtonsProps {
  sessionStepID: string;
  onFeedback: (rating: 'positive' | 'negative', sessionStepID: string) => void;
}

export const FeedbackButtons: React.FC<FeedbackButtonsProps> = ({ ses
sionStepID, onFeedback }) => {
  const [selectedRating, setSelectedRating] = useState<'positive' | 'negati
ve' | null>(null);

  const handleFeedback = (rating: 'positive' | 'negative') => {
    setSelectedRating(rating);
    onFeedback(rating, sessionStepID);
  };

  return (
    <div className="feedback-buttons" style={{ display: 'flex', gap: '8p
x', marginTop: '8px' }}>
      <button
        onClick={() => handleFeedback('positive')}
        className={selectedRating === 'positive' ? 'active' : ''}
        style={{
          padding: '6px 12px',
          border: '1px solid #e0e0e0',
          borderRadius: '4px',
```

```
            background: selectedRating === 'positive' ? '#e8f5e9' : 'transp
arent',
            cursor: 'pointer',
          }}
          title="Helpful"
        >
          👍
        </button>
        <button
          onClick={() => handleFeedback('negative')}
          className={selectedRating === 'negative' ? 'active' : ''}
          style={{
            padding: '6px 12px',
            border: '1px solid #e0e0e0',
            borderRadius: '4px',
            background: selectedRating === 'negative' ? '#ffebee' : 'transp
arent',
            cursor: 'pointer',
          }}
          title="Not helpful"
        >
          👎
        </button>
      </div>
    );
};
```

Usage with SDK:

```
const { sendFeedback, messages } = useHyperFlow({...});

{messages.map((msg, i) => (
  msg.type === 'generator' && msg.sessionStepID && (
    <FeedbackButtons
      key={i}
      sessionStepID={msg.sessionStepID}
      onFeedback={sendFeedback}
    />
```

```
    )
))}
```

References Display (RAG Sources)

For showing knowledge base sources used in responses:

```
interface ReferencesProps {
  references: Array<{
    index: number;
    document?: string;
    disabled?: boolean;
    pages: Array<{
      index: number;
      page?: string;
      text?: string;
      disabled?: boolean;
      links: Array<{
        url: string;
        title: string;
        disabled?: boolean;
      }>;
      images: Array<{
        data: string;
        disabled?: boolean;
      }>;
    }>;
  }>;
}

export const ReferencesDisplay: React.FC<ReferencesProps> = ({ referenc
es }) => {
  return (
    <div className="references" style={{ marginBottom: '20px', fontSize:
'13px' }}>
      <div style={{ fontWeight: 600, marginBottom: '8px' }}>References:
</div>
      {references.map((doc, i) => (
```

```
          !doc.disabled && (
            <div key={i} style={{ marginLeft: '12px', marginBottom: '12px' }}
>
              {doc.document && (
                <div style={{ fontWeight: 500, marginBottom: '4px' }}>
                  Document: {doc.document}
                </div>
              )}
              {doc.pages.map((page, j) => (
                !page.disabled && (
                  <div key={j} style={{ marginLeft: '12px', marginBottom:
'8px' }}>
                    {page.page && (
                      <div style={{ color: '#666', fontSize: '12px' }}>
                        Page {page.page}:
                      </div>
                    )}
                    {page.links.map((link, k) => (
                      !link.disabled && (
                        <a
                          key={k}
                          href={link.url}
                          target="_blank"
                          rel="noopener noreferrer"
                          style={{
                            display: 'block',
                            color: '#437cbe',
                            textDecoration: 'underline',
                            marginLeft: '8px',
                            marginBottom: '4px',
                          }}
                        >
                          {link.title}
                        </a>
                      )
                    ))}
                    {page.images.map((img, k) => (
                      !img.disabled && (
```

```
                                <img
                                    key={k}
                                    src={img.data}
                                    alt="Reference"
                                    style={{
                                        maxWidth: '300px',
                                        borderRadius: '4px',
                                        marginLeft: '8px',
                                        marginTop: '4px',
                                    }}
                                />
                            )
                        ))}
                    </div>
                )
            ))}
        </div>
    )
))}
    </div>
  );
};
```

Usage:

```
{messages.map((msg, i) => (
  msg.type === 'references' && msg.references && (
    <ReferencesDisplay key={i} references={msg.references} />
  )
))}
```

Complete Message Component

Combining all the above patterns:

```
interface MessageProps {
  message: ChatMessage;
  onFeedback?: (rating: 'positive' | 'negative', sessionStepID: string) => voi
```

```
d;
}

export const Message: React.FC<MessageProps> = ({ message, onFeedba
ck }) ⇒ {
   const { type, text, media, references, sessionStepID } = message;

   // User message
   if (type === 'user') {
      return (
         <div className="message user">
            <div className="bubble">{text}</div>
         </div>
      );
   }

   // Assistant message (with markdown)
   if (type === 'generator' || type === 'streamedGenerator') {
      return (
         <div className="message assistant">
            <MarkdownRenderer content={text || ''} />
            {type === 'generator' && sessionStepID && onFeedback && (
               <FeedbackButtons sessionStepID={sessionStepID} onFeedbac
k={onFeedback} />
            )}
         </div>
      );
   }

   // System message
   if (type === 'message') {
      return (
         <div className="message system">
            <MarkdownRenderer content={text || ''} />
         </div>
      );
   }
```

```
    // Error message
    if (type === 'error') {
        return (
            <div className="message error">
                Error: {text}
            </div>
        );
    }

    // Media
    if (type === 'media' && media) {
        return <MediaRenderer media={media} />;
    }

    // References
    if (type === 'references' && references) {
        return <ReferencesDisplay references={references} />;
    }

    return null;
};
```

Usage:

```
const { messages, sendFeedback } = useHyperFlow({...});

<div className="messages">
    {messages.map((msg, i) => (
        <Message key={i} message={msg} onFeedback={sendFeedback} />
    ))}
</div>
```

This provides a clean, reusable message component that handles all HyperFlow response types.

Migration from Direct API

If you've already built an integration using the manual approach from Building Custom Apps, here's how to migrate:

Before (Direct API)

```
// Manual polling loop
async function pollProgress() {
    const response = await fetch('/api/flowgraph/control/progress', {
        method: 'POST',
        headers: { 'Content-Type': 'application/json' },
        body: JSON.stringify({
            sessionID: this.sessionID,
            currentStepIndex: this.currentStepIndex,
            progress: [{ type: 'updatePoll' }]
        })
    });

    const data = await response.json();
    this.currentStepIndex = data.data.stepIndex;

    for (const response of data.data.responses) {
        switch (response.type) {
            case 'interaction':
                this.currentInteraction = response.interactionSpec;
                showInputUI();
                break;
            case 'generatedText':
                addToChat({ type: 'bot', text: response.text });
                break;
            // ... many more cases
        }
    }

    // Immediately re-poll
    this.pollProgress();
}

// Manual message sending with paramSpec
async function sendMessage(text) {
    const progress = [{
        type: 'prompt',
```

```
      data: {
        text,
        paramSpec: this.currentInteraction  // Must remember to include
      }
    }];

    this.currentInteraction = null;  // Must remember to clear
    await pollProgress(progress);
}
```

After (SDK)

```
const { messages, sendMessage, isReady } = useHyperFlow({
    apiKey: 'hf_...',
    flowGraphID: '67ca914369e9b8a95c967fdf',
});

// That's it! SDK handles:
// - Polling loop
// - Interaction specs
// - Step index
// - paramSpec inclusion
// - Spec clearing
```

Migration Checklist

- [ ] Replace manual polling loop with `useHyperFlow` hook
- [ ] Remove `sessionID` and `currentStepIndex` state (SDK manages)
- [ ] Remove interaction spec storage (SDK manages)
- [ ] Replace manual `fetch` calls with `sendMessage()`, `uploadFile()`, etc.
- [ ] Remove manual `paramSpec` handling (SDK does it)
- [ ] Replace response processing with event callbacks
- [ ] Update TypeScript types to use SDK types
- [ ] Test thoroughly (use SDK test page)

Testing Your Integration

Using the SDK Test Page

HyperFlow provides a built-in test page for debugging SDK integrations:

URL: `https://hyperflow-ai.com/hyperflow/sdk-test`

Local Development: `http://localhost:3000/hyperflow/sdk-test`

Quick Test URL:

```
http://localhost:3000/hyperflow/sdk-test?apiKey=hf_xxx&fg=67ca914369e9b8a95c967fdf
```

Features:

- Input form for API key and flowGraphID
- Debug info display (sessionID, stepIndex, origin)
- Real-time message testing
- Network tab inspection
- Easy URL sharing for reproducing issues

Debugging Tips

1. Enable Debug Mode:

The easiest way to debug SDK issues is to enable debug logging:

```
const { messages, sendMessage } = useHyperFlow({
    apiKey: 'hf_...',
    flowGraphID: '...',
    debug: true,  // ← Logs all Control API interactions
});
```

Console output shows:

- All POST requests (URL, body as formatted JSON)
- All responses (formatted JSON)
- All errors with details

Example output:

```
[HyperFlow SDK] POST /api/flowgraph/control/start
Request: {
  "flowGraphID": "67ca914369e9b8a95c967fdf",
  "queryParams": { "userId": "user-123" }
}
Response: {
  "success": true,
  "data": {
    "sessionID": "68ecb5d0c96c227ef841b63d",
    "stepIndex": 0,
    "responses": [
      { "type": "start", "flowGraphName": "My Bot" },
      { "type": "interaction", "interactionSpec": {...} }
    ]
  }
}

[HyperFlow SDK] POST /api/flowgraph/control/progress
Request: {
  "sessionID": "68ecb5d0c96c227ef841b63d",
  "currentStepIndex": 1,
  "progress": [
    {
      "type": "prompt",
      "data": {
        "text": "Hello!",
        "paramSpec": {...}
      }
    }
  ]
}
Response: {...}
```

Important: Disable debug mode in production (performance impact from verbose logging).

2. Check Browser Console:

```
// SDK logs errors automatically
// Look for: "SDK Error:", "Failed to...", etc.
```

3. Check Network Tab:

```
Should see:
✅ One /control/progress request open (waiting)
✅ One /control/streaming request (when generating)

Should NOT see:
❌ Rapid-fire /control/progress requests (tight polling bug)
❌ Multiple concurrent /control/progress (except during generation.start)
```

3. Check Origin Header:

```
Request Headers:
Origin: https://myapp.com
X-API-Key: hf_...
Accept-Language: en

Verify origin matches your API key's allowed origins
```

4. Use React DevTools:

```
// Inspect SDK state
useHyperFlowStore.getState();
```

Common Issues

"Origin not allowed for this API key"

- Your domain isn't in the API key's allowed origins list
- Add it in Account Settings → API Configuration → Edit → Allowed Origins

"Missing required field 'paramSpec'"

- SDK bug (shouldn't happen) - report this
- Check you're using the latest SDK version

Tight polling loop (thousands of requests)

- SDK bug - check you don't have an old version with `longPollingLoop()`
- Current SDK uses reactive event-driven pattern

Messages not appearing

- Check `messages` array in React DevTools
- Verify callbacks are firing (add console.logs)
- Check for JavaScript errors in console

Reference Implementation

SimulatorBotSDK Component

Location: `src/apps/hyperflow/components/chatbots/SimulatorBotSDK.tsx`

A complete chatbot UI implementation using the SDK, demonstrating:

- ✅ All interaction types (prompt, file upload, branch choice)
- ✅ Feedback buttons (thumbs up/down)
- ✅ Media rendering (images, video, audio, SVG)
- ✅ References display (RAG sources)
- ✅ Streaming with typing animation
- ✅ Error handling
- ✅ Styled components

Usage as Reference:

```
// See how the SDK is used in production
import SimulatorBotSDK from '@hyperflow/components/chatbots/Simulato
rBotSDK';

<SimulatorBotSDK
  apiKey="hf_..."
  flowGraphID="..."
  baseURL="https://hyperflow-ai.com"
```

```
    queryParams={{ userId: '123' }}
/>
```

Compare with original `SimulatorBot.js` to see the difference between direct API usage and SDK usage.

API Reference

Complete Type Definitions

See: `src/sdk/client/types.ts` for complete TypeScript definitions

Key Types:

```
// Configuration
interface HyperFlowClientConfig {
    baseURL: string;
    apiKey: string;
    flowGraphID: string;
    queryParams?: Record<string, any>;
    language?: string;
}

// Messages
interface ChatMessage {
    type: 'user' | 'assistant' | 'system' | 'error' | 'streamedGenerator' | 'gener
ator' | 'message' | 'media' | 'references';
    text?: string;
    media?: MediaContent;
    references?: Reference[];
    sessionStepID?: string;
    metadata?: ResponseMetadata;
    timestamp?: number;
}

// Metadata (LLM generation info)
interface ResponseMetadata {
    usage?: {
        promptTokens: number;
        completionTokens: number;
```

```
        totalTokens: number;
    };
    model?: {
        name: string;
        provider: string;
    };
    finishReason?: string;
    timing?: {
        elapsedNs: number;  // Nanoseconds
    };
    providerMetadata?: Record<string, any>;
}

// Interaction specs
interface PromptInteractionSpec {
    type: 'prompt';
    pathName: string;
    placeholder?: string;
    promptButtons?: Array<{
        label: string;
        promptText: string;
    }>;
}

interface FileUploadInteractionSpec {
    type: 'fileUpload';
    pathName: string;
    fileType?: string;
    optional?: boolean;
    orgID: string;
}

interface BranchChoiceInteractionSpec {
    type: 'branchChoice';
    pathName: string;
    branches: Array<{
        label: string;
        id: string;
```

```
    }>;
}
```

Example: Complete Production App

```
// ChatPage.tsx - Full-featured chat interface
import React, { useState, useEffect, useRef } from 'react';
import { useHyperFlow } from '@mirinae/sdk/react';
import { useAuth } from '@/hooks/useAuth';
import { useChatRepository } from '@/repositories/chat';

export default function ChatPage({ conversationId }: { conversationId: strin
g }) {
    const { user } = useAuth();
    const chatRepo = useChatRepository();
    const [conversation, setConversation] = useState(null);
    const [inputValue, setInputValue] = useState('');
    const messagesEndRef = useRef<HTMLDivElement>(null);

    // Load conversation from your database
    useEffect(() => {
        chatRepo.getConversation(conversationId).then(setConversation);
    }, [conversationId]);

    // Initialize HyperFlow with SDK
    const {
        messages,
        sendMessage,
        uploadFile,
        sendFeedback,
        isReady,
        isGenerating,
        busyMessage,
        error,
        sessionID,
        promptInteractionSpec,
        fileUploadInteractionSpec,
```

```
  } = useHyperFlow({
    baseURL: 'https://hyperflow-ai.com',
    apiKey: process.env.NEXT_PUBLIC_HYPERFLOW_API_KEY!,
    flowGraphID: process.env.NEXT_PUBLIC_HYPERFLOW_FLOWGRAPH_I
D!,
    queryParams: {
      userId: user.id,
      conversationId,
      language: user.language || 'en',
    },
    resumeFromSessionID: conversation?.hyperflowSessionID,
    autoStart: true,

    // Save HyperFlow sessionID to your database
    onStart: async (flowGraphName) => {
      if (sessionID) {
        await chatRepo.updateSession(conversationId, sessionID);
      }
    },

    // Save assistant messages to your database
    onGeneratedText: async (text, metadata) => {
      await chatRepo.saveMessage({
        conversationId,
        role: 'assistant',
        content: text,
        tokens: metadata?.usage?.totalTokens,
        model: metadata?.model?.name,
        timestamp: new Date(),
      });

      // Track usage for billing
      await chatRepo.recordUsage({
        userId: user.id,
        tokens: metadata?.usage?.totalTokens || 0,
      });
    },
```

```
    // Auto-generated chat titles
    onTitleUpdate: async (title) => {
      await chatRepo.updateTitle(conversationId, title);
      setConversation(prev => ({ ...prev, title }));
    },

    onError: (error) => {
      console.error('HyperFlow error:', error);
      // Show user-friendly error
      showToast('An error occurred. Please try again.');
    },
  });

  // Auto-scroll to bottom
  useEffect(() => {
    messagesEndRef.current?.scrollIntoView({ behavior: 'smooth' });
  }, [messages]);

  const handleSend = async () => {
    if (!inputValue.trim() || !isReady || isGenerating) return;

    const messageText = inputValue;
    setInputValue('');

    try {
      // Save user message to your database
      await chatRepo.saveMessage({
        conversationId,
        role: 'user',
        content: messageText,
        timestamp: new Date(),
      });

      // Send to HyperFlow
      await sendMessage(messageText);
    } catch (err) {
      console.error('Failed to send:', err);
      setInputValue(messageText); // Restore on error
```

```
      }
    };

    const handleFileUpload = async (e: React.ChangeEvent<HTMLInputElem
ent>) => {
      const file = e.target.files?.[0];
      if (!file || !fileUploadInteractionSpec) return;

      // Upload to HyperFlow
      const formData = new FormData();
      formData.append('org_id', fileUploadInteractionSpec.orgID);
      formData.append('file', file);

      const response = await fetch('https://hyperflow-ai.com/api/hyperflow/
uploadContent', {
        method: 'POST',
        body: formData,
      });

      const { data } = await response.json();
      await uploadFile(file, data.uploads, data.filename);
    };

    return (
      <div className="chat-container">
        {/* Header */}
        <div className="chat-header">
          <h1>{conversation?.title || 'Chat'}</h1>
          <div className="status">
            {error ? (
              <span className="error">Error: {error.message}</span>
            ) : isGenerating ? (
              <span className="generating">{busyMessage || 'Generatin
g...'}</span>
            ) : isReady ? (
              <span className="ready">Ready</span>
            ) : (
              <span className="connecting">Connecting...</span>
```

```
            )}
          </div>
        </div>

        {/* Messages */}
        <div className="messages">
          {messages.map((msg, i) => (
            <div key={i} className={`message ${msg.type}`}>
              {msg.type === 'user' && (
                <div className="user-message">{msg.text}</div>
              )}
              {(msg.type === 'generator' || msg.type === 'streamedGener
ator') && (
                <div className="assistant-message">
                  <div className="text">{msg.text}</div>
                  {msg.type === 'generator' && msg.sessionStepID && (
                    <div className="feedback">
                      <button onClick={() => sendFeedback('positive', ms
g.sessionStepID)}>
                        👍 Helpful
                      </button>
                      <button onClick={() => sendFeedback('negative', m
sg.sessionStepID)}>
                        👎 Not Helpful
                      </button>
                    </div>
                  )}
                </div>
              )}
              {msg.type === 'error' && (
                <div className="error-message">{msg.text}</div>
              )}
            </div>
          ))}
          <div ref={messagesEndRef} />
        </div>

        {/* Input */}
```

```
        <div className="input-container">
          {fileUploadInteractionSpec && (
            <input
              type="file"
              onChange={handleFileUpload}
              accept={fileUploadInteractionSpec.fileType || '*/*'}
            />
          )}
          <input
            type="text"
            value={inputValue}
            onChange={e => setInputValue(e.target.value)}
            onKeyPress={e => e.key === 'Enter' && !e.shiftKey && handleSe
nd()}
            placeholder={promptInteractionSpec?.placeholder || 'Type a m
essage...'}
            disabled={!isReady || isGenerating}
          />
          <button onClick={handleSend} disabled={!isReady || isGeneratin
g || !inputValue.trim()}>
            {isGenerating ? 'Sending...' : 'Send'}
          </button>
        </div>
      </div>
    );
}
```

Related Documentation

Getting Started

- Introduction to HyperFlow for Developers - Platform overview
- Building Custom Apps (Manual Approach) - Direct Control API usage

Control API Reference

- Control API Reference - Complete API documentation
- Interaction Guide - Handling user interactions

- Streaming API - LLM streaming details

Security

- API Key Management - Managing API keys
- Origin Restrictions - Client-side security

Examples

- **SDK Test Page**: `/hyperflow/sdk-test` (interactive testing)
- **Reference Component**: `src/apps/hyperflow/components/chatbots/SimulatorBotSDK.tsx`
- **Simple Example**: `src/sdk/examples/SimpleChatExample.tsx`
- **Headless Example**: `src/sdk/examples/headless-example.ts`

Support

Getting Help

- **Documentation**: Start with this guide and the Control API reference
- **Discord**: HyperFlow Support Server
- **Email**: team@hyperflow-ai.com
- **GitHub**: Report SDK issues or request features

Common Questions

Q: Can I use the SDK server-side (Node.js)?
A: Yes! The core SDK is framework-agnostic. Just initialize `HyperFlowClient` in your Node.js code.

Q: Does the SDK work with Next.js/Vue/Angular?
A: The core SDK works with any framework. React hooks are React-specific. For other frameworks, use the headless `HyperFlowClient` and integrate with your framework's state management.

Q: Can I customize the UI completely?
A: Yes! Use the headless `HyperFlowClient` and build your own UI. The SDK only handles API communication.

Q: How do I handle authentication in my app?
A: Use origin-restricted API keys for client-side apps. For server-side

integrations, use unrestricted API keys and implement your own authentication layer.

Q: Can I use multiple flow-graphs in one app?
A: Yes! Create multiple SDK instances, each with its own `flowGraphID`.

Q: How do I migrate from the manual Control API approach?
A: See the "Migration from Direct API" section above. The SDK can coexist with manual code during gradual migration.

Q: What about rate limits?
A: API keys have usage quotas. Check your organization's plan limits in Account Settings → Usage.

Changelog

v1.0 (October 2025) - Initial release

- Core headless SDK (`HyperFlowClient`)
- React integration (`useHyperFlow` hook)
- Zustand-based state management
- Complete type definitions
- Production-tested patterns
- Origin-restricted API key support
- SimulatorBotSDK reference implementation
- SDK test page for debugging

Last Updated: October 2025
Maintainers: HyperFlow Team
Feedback: team@hyperflow-ai.com

File Upload Handling with HyperFlow SDK

This guide shows how to handle file uploads in your HyperFlow-powered application, including displaying uploaded files and accessing them via URLs.

Overview

HyperFlow supports file uploads through the Control API. When a flow-graph requires file input, it returns a `fileUpload` interaction spec. The SDK provides:

1. `uploadFile()` method to send files
2. `getFileURL()` helper to generate download/display URLs
3. Automatic URL generation in chat history

Basic File Upload Flow

```
import { useHyperFlow } from '@hyperflow/client-sdk/react';

function ChatBot() {
    const {
        fileUploadInteractionSpec,
        uploadFile,
        getFileURL,
        messages
    } = useHyperFlow({
        baseURL: 'https://hyperflow-ai.com',
        apiKey: 'hf_...',
        flowGraphID: 'your-flow-graph-id'
    });

    const handleFileSelect = async (event) => {
        const file = event.target.files[0];
        if (!file) return;

        // Upload to HyperFlow (returns file ID in uploads array)
        const formData = new FormData();
        formData.append('file', file);

        const uploadResponse = await fetch(`${baseURL}/api/hyperflow/file/u
pload`, {
            method: 'POST',
            headers: { 'X-API-Key': apiKey },
            body: formData
        });

        const { fileID } = await uploadResponse.json();
```

```
    // Send to flow-graph
    await uploadFile(file, [{ data: fileID }], file.name);
  };

  return (
    <div>
      {fileUploadInteractionSpec && (
        <input type="file" onChange={handleFileSelect} />
      )}
    </div>
  );
}
```

Displaying Uploaded Files

Option 1: Using Chat History (Automatic)

The SDK automatically adds uploaded files to chat history with URLs:

```
function MessageList({ messages, getFileURL }) {
  return (
    <div>
      {messages.map((msg, idx) => {
        if (msg.type === 'user' && msg.uploads) {
          return (
            <div key={idx}>
              <p>{msg.text}</p>
              {msg.uploads.map((upload, i) => (
                <div key={i}>
                  {upload.mimetype?.startsWith('image/') ? (
                    <img src={upload.url} alt={upload.fileName} />
                  ) : (
                    <a href={upload.url} download>
                      Download {upload.fileName}
                    </a>
                  )}
                </div>
              ))}
```

```
                </div>
              );
            }
            // ... other message types
          })}
        </div>
    );
}

function ChatBot() {
    const { messages, getFileURL } = useHyperFlow({...});
    return <MessageList messages={messages} getFileURL={getFileURL} /
>;
}
```

Option 2: Using getFileURL() Helper

For custom handling or non-chat displays:

```
function FileGallery({ uploads, getFileURL }) {
    return (
        <div className="gallery">
            {uploads.map((upload) => {
                const url = getFileURL(upload.data);
                return (
                    <div key={upload.data}>
                        <img src={url} alt={upload.fileName} />
                        <a href={url} download={upload.fileName}>
                            Download
                        </a>
                    </div>
                );
            })}
        </div>
    );
}
```

Complete Example: Image Upload with Preview

```
import { useHyperFlow } from '@hyperflow/client-sdk/react';
import { useState } from 'react';

function ImageChatBot() {
    const {
        messages,
        fileUploadInteractionSpec,
        uploadFile,
        sendMessage,
        isReady,
        getFileURL
    } = useHyperFlow({
        baseURL: 'https://hyperflow-ai.com',
        apiKey: 'hf_...',
        flowGraphID: 'image-analysis-bot'
    });

    const [previewURL, setPreviewURL] = useState(null);

    const handleFileSelect = async (event) => {
        const file = event.target.files[0];
        if (!file || !file.type.startsWith('image/')) {
            alert('Please select an image file');
            return;
        }

        // Show local preview
        const localURL = URL.createObjectURL(file);
        setPreviewURL(localURL);

        try {
            // Upload to HyperFlow
            const formData = new FormData();
            formData.append('file', file);

            const response = await fetch(
                `${baseURL}/api/hyperflow/file/upload`,
                {
```

```
            method: 'POST',
            headers: { 'X-API-Key': apiKey },
            body: formData
        }
      );

      const { fileID } = await response.json();

      // Send to flow-graph
      // SDK will automatically add URL to chat history
      await uploadFile(file, [{
        data: fileID,
        fileName: file.name,
        mimetype: file.type,
        size: file.size
      }], file.name);

    } catch (error) {
      console.error('Upload failed:', error);
      alert('Failed to upload file');
    } finally {
      // Clean up preview
      URL.revokeObjectURL(localURL);
      setPreviewURL(null);
    }
  };

  return (
    <div className="chat-container">
      {/* Messages */}
      <div className="messages">
        {messages.map((msg, idx) => (
          <div key={idx} className={`message ${msg.type}`}>
            {msg.text && <p>{msg.text}</p>}

            {/* Display uploaded images */}
            {msg.uploads && msg.uploads.map((upload, i) => (
              <div key={i} className="upload-preview">
```

```
                {upload.mimetype?.startsWith('image/') ? (
                  <img
                    src={upload.url}
                    alt={upload.fileName}
                    style={{ maxWidth: '300px' }}
                  />
                ) : (
                  <a href={upload.url} download={upload.fileName}>
                    Download {upload.fileName}
                  </a>
                )}
              </div>
            ))}

            {/* Display media responses from LLM */}
            {msg.media && (
              <img src={msg.media.data} alt="Generated image" />
            )}
          </div>
        ))}
      </div>

      {/* File upload input */}
      {fileUploadInteractionSpec && (
        <div className="file-input">
          {previewURL && (
            <img
              src={previewURL}
              alt="Preview"
              style={{ maxWidth: '200px' }}
            />
          )}
          <input
            type="file"
            accept="image/*"
            onChange={handleFileSelect}
          />
        </div>
```

```
        )}

        {/* Text input */}
        {isReady && (
          <input
            type="text"
            placeholder="Type a message..."
            onKeyPress={(e) => {
              if (e.key === 'Enter') {
                sendMessage(e.currentTarget.value);
                e.currentTarget.value = '';
              }
            }}
          />
        )}
      </div>
    );
}
```

Optional vs Required Uploads

The SDK handles both upload types automatically:

Required Upload (uploaded immediately)

```
// Flow-graph has required file upload
// User must upload before proceeding

await uploadFile(file, uploads, fileName);
// → Sent to server immediately
// → Flow-graph continues with the file
```

Optional Upload (sent with next prompt)

```
// Flow-graph has optional file upload
// User can upload OR just send text

// User uploads file
```

```
await uploadFile(file, uploads, fileName);
// → Held internally (not sent yet)

// User sends text
await sendMessage("What's in this image?");
// → File + text sent together
// → Flow-graph receives both
```

Upload Object Structure

When you call `uploadFile()`, the uploads array should contain:

```
[{
   data: 'file-id-from-upload-endpoint',  // Required
   fileName: 'photo.jpg',                 // Optional
   mimetype: 'image/jpeg',                // Optional
   size: 123456                           // Optional
}]
```

After processing, chat history entries will include:

```
{
   type: 'user',
   text: 'Uploaded: photo.jpg',
   uploads: [{
      data: 'file-id-from-upload-endpoint',
      fileName: 'photo.jpg',
      mimetype: 'image/jpeg',
      size: 123456,
      url: 'https://hyperflow-ai.com/api/hyperflow/file/get/media/file-id-from
-upload-endpoint'
   }],
   timestamp: 1234567890
}
```

Direct URL Access

For cases where you need the URL outside of chat history:

```
const { getFileURL } = useHyperFlow({...});

// Generate URL from file ID
const downloadURL = getFileURL('file-id-123');
// → 'https://hyperflow-ai.com/api/hyperflow/file/get/media/file-id-123'

// Use in your UI
<a href={downloadURL} download>Download</a>
<img src={downloadURL} />
<video src={downloadURL} controls />
```

Session Restoration with File URLs

When restoring a session with `initialMessages`, file URLs are preserved:

```
// Load from localStorage
const savedMessages = JSON.parse(localStorage.getItem('chat-history'));

const { messages } = useHyperFlow({
  baseURL: 'https://hyperflow-ai.com',
  apiKey: 'hf_...',
  flowGraphID: 'bot-id',
  initialMessages: savedMessages,  // Includes upload URLs
  resumeFromSessionID: savedSessionID
});

// Messages with uploads will have URLs ready to use
// No need to regenerate them
```

API Reference

`getFileURL(fileId: string): string`

Generates the full URL to access an uploaded file.

Parameters:

- `fileId` - File ID from the upload response (found in `upload.data`)

Returns:

- Full URL string in format: `{baseURL}/api/hyperflow/file/get/media/{fileId}`

Example:

```
const url = getFileURL('67abc123def456');
// → 'https://hyperflow-ai.com/api/hyperflow/file/get/media/67abc123def456'
```

Notes

- URLs are automatically added to chat history entries for uploads
- The `getFileURL()` helper uses the `baseURL` from your SDK configuration
- File access requires the same API key used to upload the file
- Files are stored in GridFS on the server
- Uploaded files are associated with the organization that owns the flow-graph

@hyperflow/client-sdk README

Official JavaScript/TypeScript SDK for building custom applications powered by HyperFlow flow-graphs.

Build AI chatbots, document analyzers, and custom workflows with minimal code. The SDK handles all Control API complexity—state management, streaming, interaction specs, and session lifecycle—letting you focus on your application's UI and business logic.

Version: 1.1.1

Features

- **Headless Core SDK** - Framework-agnostic Control API client
- **React Hooks** - `useHyperFlow` hook with state management
- **TypeScript** - Full type definitions included
- **Streaming** - Automatic LLM streaming with typing animation

- **State Management** - Handles interaction specs, step indices, session lifecycle
- **Session Continuity** - True session resumption with same sessionID (v1.1)
- **File Upload Helpers** - Auto-generate URLs for uploaded files (v1.1)
- **Debug Mode** - Verbose logging for development
- **Production-Tested** - Based on HyperFlow's own chatbot implementations

Installation

Current: Local/Tarball Install

```
# From tarball
yarn add file:./hyperflow-client-sdk-v1.1.0.tgz

# Or from local directory (during development)
yarn add file:/path/to/hyperflow-mono/src/sdk
```

Future: NPM Package

```
npm install @hyperflow/client-sdk
# or
yarn add @hyperflow/client-sdk
```

Quick Start

1. Get Your API Key

1. Log into HyperFlow
2. Go to **Account Settings → API Configuration**
3. Generate a new API key
4. Configure **Allowed Origins** for client-side usage:

   ```
   https://myapp.com
   http://localhost:3000
   ```

2. Get Your FlowGraph ID

Create and publish a flow-graph in HyperFlow IDE, then copy the flowGraphID (24-character hex string).

3. React Example

```
import React, { useState } from 'react';
import { useHyperFlow } from '@hyperflow/client-sdk/react';

function MyChat() {
  const [input, setInput] = useState('');

  const {
    messages,
    sendMessage,
    isReady,
    isGenerating,
    error,
  } = useHyperFlow({
    baseURL: 'https://hyperflow-ai.com',
    apiKey: 'hf_your_api_key',
    flowGraphID: '67ca914369e9b8a95c967fdf',
    queryParams: { userId: 'user-123' },
    debug: true,  // Enable during development

    // Optional: Save to your database
    onGeneratedText: (text, metadata) => {
      console.log('Tokens:', metadata?.usage?.totalTokens);
    },
  });

  const handleSend = async () => {
    if (!input.trim() || !isReady) return;
    await sendMessage(input);
    setInput('');
  };

  return (
```

```
      <div>
        {/* Messages */}
        {messages.map((msg, i) ⇒ (
          <div key={i}>
            <strong>{msg.type === 'user' ? 'You' : 'Bot'}:</strong>
            {msg.text}
          </div>
        ))}

        {/* Status */}
        {error && <div>Error: {error.message}</div>}
        {isGenerating && <div>Generating...</div>}

        {/* Input */}
        <input
          value={input}
          onChange={e ⇒ setInput(e.target.value)}
          onKeyPress={e ⇒ e.key === 'Enter' && handleSend()}
          disabled={!isReady || isGenerating}
        />
        <button onClick={handleSend} disabled={!isReady || isGenerating}>
          Send
        </button>
      </div>
    );
}
```

4. Headless Client (Vanilla JS)

```
import { HyperFlowClient } from '@hyperflow/client-sdk/client';

const client = new HyperFlowClient({
  baseURL: 'https://hyperflow-ai.com',
  apiKey: 'hf_...',
  flowGraphID: '67ca914369e9b8a95c967fdf',
  queryParams: { userId: 'user-123' },
  debug: true,  // Enable during development
});
```

```
// Event handlers
client.on('generatedText', (text, metadata) ⇒ {
   console.log('Bot:', text);
   console.log('Tokens:', metadata?.usage?.totalTokens);
});

client.on('interaction', (spec) ⇒ {
   if (spec.type === 'prompt') {
      enableUserInput();
   }
});

client.on('error', (error) ⇒ {
   console.error('Error:', error);
});

// Start and interact
await client.start();
await client.sendMessage('Hello!');
await client.end();
```

Configuration

```
interface HyperFlowClientConfig {
   baseURL: string;          // HyperFlow API URL
   apiKey: string;           // Your API key (hf_...)
   flowGraphID: string;      // Published flow-graph ID
   queryParams?: object;     // Context data for flow
   language?: string;        // Default: 'en'
   debug?: boolean;          // Enable verbose logging
   initialMessages?: ChatMessage[];  // Seed messages (for UI restoration)
}
```

Debug Mode:

```
useHyperFlow({
    ...config,
    debug: true,  // Logs all API requests/responses
});
```

Initial Messages (for session restoration):

```
const saved = JSON.parse(localStorage.getItem('chat') || '[]');

useHyperFlow({
    ...config,
    initialMessages: saved,  // Populates messages array on mount
});
```

Common Patterns

Session Restore

⚠️ **Important**: Two separate concerns when resuming:

- `resumeFromSessionID` → Restores LLM context (uses /control/refresh to continue same session)
- `initialMessages` → Restores UI messages

Complete restoration pattern:

```
const storageKey = `chat-${conversationId}`;

// Load saved data
const savedMessages = JSON.parse(localStorage.getItem(storageKey) ||
'[]');
const savedSessionID = localStorage.getItem(`session-${conversationId}`);

const { messages, sessionID } = useHyperFlow({
    resumeFromSessionID: savedSessionID,     // Restores LLM context
    initialMessages: savedMessages,          // Restores UI messages
});
```

```
// Auto-save (standard React pattern)
useEffect(() => {
    localStorage.setItem(storageKey, JSON.stringify(messages));
    if (sessionID) {
        localStorage.setItem(`session-${conversationId}`, sessionID);
    }
}, [messages, sessionID]);

// Single source of truth - SDK messages array
{messages.map(msg => <div>{msg.text}</div>)}
```

What each part does:

- `resumeFromSessionID` → Restores conversation history for AI ✅
- `initialMessages` → Populates SDK messages array for UI ✅
- `useEffect` → Persists messages (your choice of storage) ✅

Storage options: localStorage, IndexedDB, your database, sessionStorage, etc.

See full guide for complete patterns.

Start New Chat

```
const { reset, start } = useHyperFlow({...});

// Option 1: Reset and restart (simplest - no page reload)
const handleNewChat = async () => {
    reset();          // Clear SDK state
    await start();    // Start fresh (auto-ends old session if needed)
};

<button onClick={handleNewChat}>New Chat</button>

// Option 2: Component remount (preserves parent state)
const [chatKey, setChatKey] = useState(0);
<button onClick={() => setChatKey(k => k + 1)}>New Chat</button>
<Chat key={chatKey} />
```

Note: `start()` automatically ends any existing session before starting a new one (unless you're resuming).

Update Query Parameters

Change context mid-session:

```
const { updateQueryParams } = useHyperFlow({
    queryParams: { userId: 'user-123' },
});

// Update conversation context
updateQueryParams({ conversationId: 'conv-456' });

// Update user preferences
updateQueryParams({ language: 'es', theme: 'dark' });

// Next progress call includes merged params
```

Use for:

- Switching conversations
- Updating user preferences
- Dynamic feature flags
- A/B testing

File Uploads

```
const { uploadFile, fileUploadInteractionSpec, getFileURL } = useHyperFlo
w({...});

const handleFileUpload = async (file: File) => {
    // 1. Upload to HyperFlow
    const formData = new FormData();
    formData.append('org_id', fileUploadInteractionSpec.orgID);
    formData.append('file', file);

    const res = await fetch('https://hyperflow-ai.com/api/hyperflow/uploadC
ontent', {
        method: 'POST',
        body: formData,
    });
```

```
  const { uploads, filename } = await res.json();

  // 2. Send to SDK
  await uploadFile(file, uploads, filename);
  // URLs automatically added to chat history
};

// Display uploaded files
{messages.map(msg ⇒
  msg.uploads?.map(upload ⇒ (
    upload.mimetype?.startsWith('image/') ? (
      <img src={upload.url} alt={upload.fileName} />
    ) : (
      <a href={getFileURL(upload.data)} download>
        Download {upload.fileName}
      </a>
    )
  ))
)}
```

Feedback & Metadata

```
const { sendFeedback } = useHyperFlow({
  onGeneratedText: (text, metadata) ⇒ {
    // Track token usage
    console.log('Tokens:', metadata?.usage?.totalTokens);
    console.log('Model:', metadata?.model?.name);
    console.log('Time:', metadata?.timing?.elapsedNs / 1_000_000, 'ms');
  },
});

// Feedback buttons
{messages.map((msg) ⇒
  msg.type === 'generator' && msg.sessionStepID && (
    <>
      <button onClick={() ⇒ sendFeedback('positive', msg.sessionStepI
D)}>👍</button>
```

```
          <button onClick={() => sendFeedback('negative', msg.sessionStepI
D)}>👎</button>
        </>
    )
)}
```

Security

Origin-Restricted API Keys

For client-side usage, configure allowed origins:

1. Go to **HyperFlow → Account Settings → API Configuration**
2. Generate new API key
3. Add **Allowed Origins**:

```
https://myapp.com
http://localhost:3000
```

Even if exposed in client code, the key only works from your approved domains.

Best Practices:

- Use HTTPS in production
- List all subdomains explicitly
- Separate keys for dev/production
- Don't leave origins empty for client keys

API Reference

React Hook

```
const {
    // State
    messages,          // Chat history
    isReady,           // Can send message?
    isGenerating,      // LLM generating?
```

```
  sessionID,        // Current session
  error,            // Last error

  // Actions
  sendMessage,      // (text) => Promise<void>
  uploadFile,       // (file, uploads, fileName) => Promise<void>
  selectBranch,     // (index) => Promise<void>
  sendFeedback,     // (rating, sessionStepID) => Promise<void>

  // Utilities
  start,            // (options?) => Promise<void> - Start/resume session
  updateQueryParams, // (params: object) => void - Update params mid-se
ssion
  getFileURL,       // (fileId) => string - Generate URL for uploaded file
  end,              // () => Promise<void> - End current session
  reset,            // () => void - Clear all state
  clearHistory,     // () => void - Clear messages only
} = useHyperFlow(config);
```

Core Client

```
class HyperFlowClient {
  // Session
  async start(options?: { resumeFromSessionID?: string }): Promise<void>
  async end(): Promise<void>

  // Actions
  async sendMessage(text: string): Promise<void>
  async uploadFile(file: File, uploads: any[], fileName: string): Promise<voi
d>
  async selectBranch(edgeIndex: number): Promise<void>
  async sendFeedback(rating: 'positive' | 'negative', sessionStepID?: strin
g): Promise<void>

  // Utilities
  updateQueryParams(newParams: Record<string, any>): void
  getFileURL(fileId: string): string  // Generate URL for uploaded file
```

```
    // Events
    on(event: string, handler: Function): void
    off(event: string, handler: Function): void

    // State
    get sessionID(): string | null
    get isReady(): boolean
    get chatHistory(): ChatMessage[]
}
```

Lifecycle Notes:

- `start()` automatically ends existing session before starting new one (unless resuming)
- `start({ resumeFromSessionID })` uses /control/refresh endpoint (preserves sessionID)
- `end()` is safe to call anytime (no-op if no session exists)
- Both methods include debug logging when `debug: true`

Troubleshooting

"Origin not allowed for this API key"

- Add your domain to API key's allowed origins

Tight polling loop (thousands of requests)

- Ensure you're using SDK v1.0+ (uses reactive pattern)

Messages not appearing

- Enable debug mode: `debug: true`
- Check browser console for errors

"Missing required field 'paramSpec'"

- SDK should handle this automatically - check version

Documentation

Complete Guides:

- Client SDK Guide - Full documentation

- Building Custom Apps - Manual Control API approach
- Control API Reference - API specification

Examples:

- Test Page: `http://localhost:3000/hyperflow/sdk-test`
- Reference Component: `src/apps/hyperflow/components/chatbots/SimulatorBotSDK.tsx`

Support

- **Documentation**: https://docs.hyperflow-ai.com
- **Discord**: https://discord.gg/hyperflow
- **Email**: team@hyperflow-ai.com
- **Issues**: https://github.com/mirinae/hyperflow-mono/issues

License