DNotifier SDK cheat sheet

All topics · JavaScript / TypeScript · Python and Dart use the same methods (snake_case in Python).

Get started

Create an app at app.dnotifier.com, copy the appId and secret, install the SDK and connect. Add model keys in AI Studio → Providers.

Install the SDK

One package per language. Node.js also needs ws for WebSocket.

```
npm install @dnotifier-realtime/dnotifier ws
```

Store your keys

Keep the secret in environment variables. Never commit it.

```
# .env
DNOTIFIER_APP_ID=your-app-id
DNOTIFIER_SECRET=your-secret
```

Connect

Use "ws" for live messages, "http" for AI, search and workflows.

```
import { DNotifier } from "@dnotifier-
realtime/dnotifier";

const notifier = new DNotifier({
  appId: process.env.DNOTIFIER_APP_ID,
  secret: process.env.DNOTIFIER_SECRET,
  transport: "http",
  userId: "user-123",
});

await notifier.connect();
```

Check your plan

See quotas, and whether AI is on, before you call it.

```
notifier.getPlanLimits();
// { aiEnabled, messagesHardLimit,
maxAIRequestsPerMonth, ... }

if (!notifier.aiEnabled) {
  throw new Error("AI is not enabled for this
plan");
}
```

Realtime messaging

Send messages straight to a userId over WebSocket. Text, JSON, images, audio and raw bytes.

Send to one user

Delivered live to the receiver. History is saved by default.

```
await notifier.send({
  senderId: "user-alice",
  receiverId: "user-bob",
  data: { type: "text", text: "Hello from
DNotifier!" },
});
```

Receive messages

Your onMessage handler fires for every incoming message.

```
onMessage: (msg) => {
  console.log(msg.metadata.sender);
  console.log(msg.payload.toJSON());
},
```

Send to many users

Pass a list of userIds. Offline users miss live delivery.

```
await notifier.send({
  senderId: "svc-notifications",
  receiverIds: ["employee-101", "employee-102"],
  data: { type: "notification", title: "Deploy
complete" },
  saveHistory: false,
});
```

Files and raw bytes

Images and audio as structured data, or raw bytes with sendBinary (ws only).

```
await notifier.send({
  senderId, receiverId,
  data: { type: "image", content: bytes, fileName,
mimeType },
});

notifier.sendBinary({
  senderId, receiverIds: ["user-bob"],
  buffer: new Uint8Array([0, 1, 2]), type: "binary",
});
// receive: msg.payload.raw()
```

Chat history

Load a conversation, or delete one message by id.

```
await notifier.fetchChatHistory({
  senderId: "user-alice",
  receiverIds: ["user-bob"],
});

await notifier.deleteChatHistoryMessage({ senderId,
messageId });
```

AI

Call any of 8 model providers through one method, keep multi-turn sessions and store history.

Send a prompt

Use the http transport so the answer comes back from the call.

```
const res = await notifier.sendAI({
  senderId: "user-123",
  message: { text: "Summarize our refund policy in
two sentences." },
});

console.log(res?.data?.content);
```

Pick provider and model

open_ai, anthropic, google_gemini, hugging_face, perplexity, azure_ai, aws_bedrock, ollama.

```
await notifier.sendAI({
  senderId: USER_ID,
  provider: "open_ai",
  model: "gpt-4o",
  message: {
    useKnowledgeBase: false,
    messages: [{ role: "user", content: "Reply with
one word: ok" }],
  },
  saveHistory: true,
});
```

Multi-turn session

Pass the first reply's id back as sessionId to keep context.

```
const first = await notifier.sendAI({
  senderId, message: { text: "Help with order
#4821." },
});
const sessionId = first?.metadata?.packet?.id;

await notifier.sendAI({
  senderId, sessionId,
  message: { text: "What documents do I need?" },
});
```

AI history

Fetch or delete a user's AI messages. Turn on logs: true to see sessions in Dashboard → Logs.

```
await notifier.fetchAIHistory({ senderId });
await notifier.deleteAIHistoryMessage({ senderId,
messageId });

// skip saving one call
await notifier.sendAI({ senderId, message,
saveHistory: false });
```

Knowledge base (RAG)

Store documents and search them by meaning. The vector index is managed for you.

Add a document

recordId is your own id. Put source and tags in metadata.

```
await notifier.addDocument({
  senderId: USER_ID,
  recordId: "refund-policy",
  content: "Refunds are processed within 5 business
days.",
  type: "text",
  metadata: { source: "help-center", locale: "en" },
});
```

Semantic search

Find the closest passages. Filter by the source you set in metadata.

```
const hits = await notifier.search({
  senderId: USER_ID,
  query: "how long do refunds take",
  limit: 5,
  minSimilarity: 0.7,
  filterbySource: "help-center",
});
```

Update, read, list, delete

Same arguments as addDocument; list is paginated.

```
await notifier.updateDocument({ senderId, recordId,
content, type, metadata });
await notifier.getDocument({ senderId, recordId });
await notifier.listDocuments({ senderId, limit: 20,
offset: 0 });
await notifier.deleteDocument({ senderId, recordId
});
```

Agents & tools

Give agents your own functions or any MCP server's tools, with permissions checked on the backend.

Register a tool

A name, a JSON schema for the input, and the function to run.

```
const add = notifier.tools.register({
  name: "math.add",
  description: "Add two numbers",
  inputSchema: {
    type: "object",
    properties: { a: { type: "number" }, b: { type:
"number" } },
    required: ["a", "b"],
  },
  async execute(input) {
    return { sum: Number(input.a) + Number(input.b)
};
  },
});
```

Define and run an agent

The agent decides when to call its tools, up to maxToolRounds.

```
const agent = DNotifier.defineAgent({
  name: "math-assistant",
  instructions: "Use math.add when the user asks to
add numbers.",
  provider: "open_ai",
  model: "gpt-4o",
  tools: [add],
  permissions: { "math.add": ["execute"] },
});

const result = await notifier.runAgent(agent, {
  input: "What is 21 + 21?",
  maxToolRounds: 6,
});
console.log(result.output, result.finishReason);
```

Connect an MCP server

Its tools are discovered and prefixed with the server id.

```
const mermaid = await notifier.mcp.connect({
  id: "mermaid",
  name: "Agentic Mermaid",
  transport: "http",
  endpoint: "https://agentic-mermaid.dev/mcp",
});

const tools = mermaid.tools();
```

Workflows

Chain named agents in order or in parallel, share state between them, and watch each run as a graph.

Build a workflow

An entry function calls agents by name. Register the agents on the workflow.

```
const flow = new DNotifier.Workflow({
  name: "support-intent-router",
  observability: true,
  async entry(ctx) {
    const intent = await ctx.runAgent("intent-
agent");
    ctx.state.intent = intent;
    return ctx.runAgent("general-agent", { input:
ctx.input });
  },
}).registerAgents({
  "intent-agent": intentAgent,
  "general-agent": generalAgent,
});
```

Run it

Returns the result and an executionId you can open in the dashboard.

```
const run = await notifier.runWorkflow({
  workflow: intentRouterWorkflow,
  input: "How do refunds work?",
});

console.log(run.result, run.executionId);
```

Agents in parallel

runAgents starts several agents at once and returns their results in order.

```
const { results } = await ctx.runAgents([
  { name: "outline-planner" },
  { name: "keyword-researcher" },
]);
const [outline, keywords] = results;
```

Wait for a person

Pause a step until a reply arrives (ws transport), for approvals.

```
const reply = await ctx.waitForMessage({ timeoutMs:
120000 });
const body = reply.payload.toJSON();
```

Agents on other servers

Point an agent name at a receiverId. That host calls workflow.listen(notifier).

```
workflow.registerAgents({
  "formatter": formatterAgent,
  "outline-planner": { receiverId: "svc-agent-
outline" },
});
```

Observability

See every AI call, tool call and workflow step in the dashboard.

Workflow graphs

observability: true on a workflow or agent. Name steps with label.

```
new DNotifier.Workflow({ name, observability: true,
entry });

await ctx.sendAI({ message, label: "Intent
classification" });
ctx.recordStep({ label: "Saved to CRM", input,
output });
```

Session logs

logs: true on the client. Open Dashboard → Logs.

```
const notifier = new DNotifier({
  appId, secret,
  transport: "http",
  userId: "user-123",
  logs: true,
});
```

Tool events in your code

Stream agent and tool events as they happen.

```
const notifier = new DNotifier({
  appId, secret, transport: "http", userId: "user-
123",
  permissionEnforcement: "backend",
  onAgentEvent: (event) => console.log(event.type,
event.toolId),
});
```

Docs: dnotifier.gitbook.io/product-docs · Examples: github.com/dnotifier-ai · Keys: app.dnotifier.com