The Ultimate n8n Guide 👾

Overview of n8n Workflows and Nodes

What is n8n?
n8n is an open-source workflow automation tool that lets you connect various apps and services without coding. It uses a visual editor where you drag-and-drop **nodes** (also called *modules*) and connect them to create automated workflows. Each node performs a specific task, and together they function like an ETL pipeline – *Extracting* data from sources, *Transforming* it, and *Loading* it to destinationsdocs.n8n.io. Nodes pass data to each other as a stream of **items**, where each item is a JSON object (an array of objects flows through the nodes)docs.n8n.io. This consistent JSON structure makes sure all nodes can understand and process the data.

Workflow Structure:
A workflow in n8n typically starts with a **Trigger node** (which waits for an event or a schedule to start the flow) and then one or more **regular nodes** that process data. Nodes are connected via their **inputs and outputs** (called *connections*) so that the output of one node becomes the input of the next. Triggers have no input; they kick off the workflow when their condition is met (like an incoming webhook call or a timer). After a trigger fires, data moves through the sequence of connected nodes. You can branch the flow (e.g., using IF nodes for conditional paths) and even merge branches back together.

Data and JSON Items:
As data moves through the workflow, n8n represents it as JSON. Each node receives an array of items (JSON objects) and typically produces an array of items as output. For example, a node might receive an array of 10 objects and output a transformed array of 10 objects. Many nodes will automatically loop over each item in the input array and process

them individually. Some nodes can also aggregate or split items as needed (we'll cover those). Because each item is JSON, you can easily reference its fields in subsequent nodes using *expressions*. n8n's expression system lets you insert values from previous node outputs into new node parameters. For example, you might set an Email node's **Subject** to {{$node["Google Sheets"].json["title"]}} to use a value from a Google Sheets node's outputcommunity.n8n.io. Generally, you will use the syntax {{$json["fieldName"]}} to reference the current item's fields or {{$node["Node Name"].json["field"]}} to reference another node's output. This mechanism is how you "wire up" data between nodes without code.

Modules vs Nodes:
In n8n, the terms *module* and *node* are often used interchangeably. Each node is essentially a module providing certain functionality or integration. n8n comes with a large collection of built-in nodes for common functions and services, and you can install community nodes for additional integrations. Below, we'll cover all the key built-in modules/nodes – including core nodes (for logic, data manipulation, etc.) and popular app integration nodes – explaining how each one works, how to configure them (including JSON structure), and best practices for using them in workflows.

Before diving into individual nodes, remember that you can always export any n8n workflow as a JSON file. The workflow JSON contains all nodes and their configurations, plus the connections between them. This allows for easy sharing and version control of workflowsdocs.n8n.io. We will show examples of these JSON structures as we discuss the nodes, so you can see how a workflow is defined in code.

Core Trigger Nodes (Starting Workflows)

Trigger nodes are special nodes that **start** a workflow execution. They wait for an external event or a scheduled time and then emit data to the rest of the workflow. Here are some of the most widely-used trigger modules in n8n:

Webhook Trigger (Webhook node): The Webhook node allows n8n to receive HTTP requests from external services and start a workflow with the received datadocs.n8n.io. You configure a unique URL path (and method) for the webhook; when an HTTP request hits that URL, the node captures the request data (query parameters, body, headers, etc.) and launches the workflow. The Webhook node is immensely powerful – it essentially lets you turn an n8n workflow into a custom API endpoint or a callback URL for web servicesdocs.n8n.iodocs.n8n.io. For example, you could create a webhook that a form on your website calls when submitted, passing the form data into n8n for processing. The Webhook node supports all standard HTTP methods (GET, POST, etc.)docs.n8n.io. It can also send a response back to the caller: by default it responds immediately (e.g., with a 200 OK), but you can also pair it with a **Respond to Webhook** node to send custom responses after the workflow finishes processing. In JSON, a Webhook node definition looks like this:

```
 json
CopyEdit
{
  "parameters": {
    "path": "my-webhook-path",
    "method": "POST",
```

```
    "responseMode": "onReceived"
  },
  "name": "Webhook",
  "type": "n8n-nodes-base.webhook",
  "typeVersion": 1,
  "position": [0, 300]
}
```

- In this example, the webhook is listening at path /my-webhook-path for POST requests. (The full URL includes your n8n host and workflow ID automatically.) The responseMode might be "onReceived" for an immediate default response, or you could set it to require a manual response. Once this node triggers, it outputs the request data as JSON (for instance, any JSON body is under item.json).

Schedule Trigger (Cron): The Schedule Trigger (formerly known as Cron node) fires the workflow on a schedule. You can configure it with a simple preset (every X minutes/hours, daily, weekly, etc.) or a CRON expression for fine-grained scheduling. Use this trigger for workflows that should run periodically – for example, a job that runs every night at 2 AM. The JSON for a Schedule Trigger node might look like:

```
 json
CopyEdit
{
  "parameters": {
    "mode": "everyDay",
    "hour": 2
  },
  "name": "Schedule Trigger",
  "type": "n8n-nodes-base.scheduleTrigger",
  "typeVersion": 1,
  "position": [0, 100]
}
```

- This would trigger the workflow every day at 2:00 (AM). The Schedule Trigger doesn't output any specific data (it just emits an empty item to start the flow), but you can combine it with other nodes to fetch or generate data on a schedule.

- Manual Trigger: This is a utility trigger used during development. The Manual Trigger node simply allows you to manually start the workflow by clicking an "Execute Workflow" button in the editor. It doesn't listen to external events; it's only for testing or running flows on demand from within n8n. The manual trigger outputs an empty item (just {}) to get the workflow going.

- Email Trigger (IMAP): The Email Trigger node connects to an IMAP email inbox and triggers the workflow whenever a new email arrives. You configure email server credentials (IMAP host, port, etc.) and optionally filters (like only unseen emails, or from a certain sender). When a new email is detected, this node outputs the email's details (subject, sender, body, attachments, etc.) as JSON. This is useful for email-driven automations (for example, "when a customer emails support, create a

ticket").

- Other Trigger Nodes: n8n offers many other trigger modules, such as RSS Feed Trigger (checks an RSS feed for new items), Webhook Trigger (GraphQL), Clockify Trigger, Stripe Trigger, etc., for various services. If a specific app has a trigger available, you can use it to start workflows on events from that app (for example, a GitHub Trigger for new GitHub issues). If no specific trigger exists, you can often use the generic Webhook or Polling approach (some regular nodes can poll for changes). But covering each service's trigger is beyond scope – suffice to say, triggers are the entry points for workflows.

Using Webhook & Respond to Webhook: One best practice when building an API-like workflow with the Webhook node is to use the Respond to Webhook node to control the HTTP response. The Webhook node can be configured to wait until a Respond node executes. This allows your workflow to process incoming data (perhaps call other services, perform logic) and then send a custom response (like a JSON payload or a specific HTTP status) back to the original caller. The Respond to Webhook node simply takes some data (or static text) and ends the request with that response. For example, you might respond with a JSON confirmation or the results of processing. If you don't use a Respond node, the Webhook will by default send a generic response (200 OK with no body, or a default message) as soon as the workflow starts or finishes based on responseMode.

Trigger Node JSON Example: To illustrate, here's a simple JSON snippet combining a Schedule Trigger and a Webhook trigger (just as examples of configuration):

json
CopyEdit

```
"nodes": [
 {
  "parameters": {
   "mode": "everyWeek",
   "weekDay": "Monday",
   "hour": 9,
   "minute": 0
  },
  "name": "Schedule Trigger",
  "type": "n8n-nodes-base.scheduleTrigger",
  "position": [0, 100]
 },
 {
  "parameters": {
   "path": "new-order",
   "methods": ["POST"]
  },
  "name": "Webhook",
  "type": "n8n-nodes-base.webhook",
  "position": [0, 300]
 }
],
"connections": {}
```

In an actual workflow, you'd typically use one trigger or the other to start the flow. The JSON structure shows the distinct node configurations. In the Schedule Trigger, mode: "everyWeek" with weekDay: "Monday" etc. sets a weekly schedule. In the Webhook, methods array allows multiple HTTP methods (here just POST) and a custom path is defined.

Core Nodes for Logic, Data, and Flow Control

Once a workflow is triggered, a variety of core nodes are available to process the data. These include nodes for HTTP calls, coding/scripting, conditional logic, data transformation, and more. We'll cover the most important ones in detail:

HTTP Request Node

The **HTTP Request** node is one of the most versatile and frequently used modules in n8n. It allows your workflow to make outbound HTTP calls to any API or URLdocs.n8n.io. Essentially, this node lets n8n interact with external web services – to GET data, POST data, update resources, etc. If n8n doesn't have a pre-built integration for a service, you can often use an HTTP Request node to call that service's REST API directlydocs.n8n.io.

Functionality: With HTTP Request, you can configure the **method** (GET, POST, PUT, DELETE, etc.)docs.n8n.io, the URL endpoint, query parameters, headers, request body, and authentication. The node can both send data (like passing along item data in the request) and receive data (parsing the response). It supports various content types (JSON, form-data, raw text, files). For example, you might use this node to GET data from an external API and then use subsequent nodes to process that data.

Authentication: The HTTP node supports multiple auth methods out of the box. In the node's parameters you'll find an **Authentication** field. You can either use **Predefined Credential** types (for services n8n knows, like OAuth2 for certain APIs) or **Generic** auth methods (Basic Auth, Bearer Token, OAuth2, etc.)docs.n8n.iodocs.n8n.io. Often, the simplest method for custom APIs is to use *Header Auth* or *Query Auth* by adding an API key as a header or query param. You can create an n8n Credential for the API (with your token, key, etc.) and select it in the HTTP Request node, which is a best practice so you don't hard-code secrets in the workflow.

Node Parameters: Some important settings of HTTP Request node include:

- **URL:** The endpoint you want to call (could be static or an expression from previous node data)docs.n8n.io.
- **Method:** GET/POST/etc.
- **Query Parameters:** You can add query params either as fields or as a raw JSON object.

- Headers: Similarly, add custom headers if needed (like Content-Type, authorization tokens, etc.).

- Body: If method supports a body (POST/PUT/PATCH), you choose the content type. The node lets you send JSON, form data (including file upload), or raw text. You can construct the body by adding fields or raw JSON.

- Options: Under "Options", you find advanced settings like "Full Response" (if you want status code and headers along with the body), "Trust SSL" flags, and "Continue On Fail" (which, if true, will not stop the workflow even if the HTTP call returns an error or non-2xx status).

Example Usage: Suppose you want to fetch weather data from a public API. You'd add an HTTP Request node, set Method = GET, URL = e.g. https://api.weatherapi.com/v1/current.json?key=<YOURKEY>&q=London. You'd configure the authentication (maybe an API key in the URL as shown). When executed, this node will output the response from the API as JSON. Typically, the output JSON is in item.json for each item input. If no input items, the node still runs once (with one empty input) by default. You can also feed multiple items to an HTTP node – by default it will run the request for each item (for instance, if you have an array of cities in incoming data, and use an expression in the URL, it will call the API for each city).

HTTP Node JSON example: A configured HTTP Request node in a workflow export might appear as below:

json
CopyEdit

```
{
  "parameters": {
    "method": "GET",
    "url": "https://api.example.com/data?limit=100",
    "authentication": "none",
    "responseFormat": "json",
    "options": {
      "fullResponse": false
    }
  },
  "name": "HTTP Request",
  "type": "n8n-nodes-base.httpRequest",
  "typeVersion": 1,
  "position": [400, 200],
  "credentials": {
    "httpHeaderAuth": {
      "id": "abc123",
      "name": "Example API Key"
    }
  }
}
```

In this snippet:

- authentication: "none" means no special auth method was selected (perhaps the API expects a header token, which we could configure under credentials as shown).

- The credentials section references a stored credential named "Example API Key" of type HTTP Header Auth (which might inject an Authorization header).

- We chose responseFormat: "json" so n8n will parse the response as JSON. (Other options include string or file/binary.)

- options.fullResponse: false indicates we only care about the body. If true, the output would include status code, headers, etc.

- Position and name are just for the editor.

The HTTP Request node is extremely useful for custom integrations. Keep in mind API rate limits when using it – if you need to call a rate-limited API many times, consider using the Wait node or the Continue On Fail option with logic to retry. (We'll discuss error handling later.)

Code (Function) Node

The Code node (formerly known as Function node) allows you to write custom code (JavaScript or Python) to manipulate data or implement custom logic within your workflow. It essentially embeds a small scripting environment into the flow. This is one of the most powerful nodes, because if the built-in nodes can't do something, you can likely achieve it with a few lines of code here.

JavaScript and Python: As of recent n8n versions, the Code node supports both Node.js JavaScript and Python. By default, it runs JavaScript, but you can toggle to Python if needed. Most users use JavaScript since n8n itself runs on Node.js and has better integration with the workflow (and more features like access to $node variables). The Code node replaced the older "Function" and "Function Item" nodes in n8n version 0.198.0docs.n8n.io (in older workflow JSON you might see nodes of type n8n-nodes-base.function; these are now consolidated into the Code node).

Usage Modes: The Code node has two modes of executiondocs.n8n.io:

- Run Once for All Items: (Default) – Your code runs a single time and can process the entire input array of items at once. In JavaScript, you'll have access to an array items which contains all input items (items[0].json is the first item's JSON data, etc.). In Python, similarly, you get an input list.

- Run Once for Each Item: – The code will execute separately for each incoming item. In JS, you then typically use a variable item (or similar) representing the current item,

rather than an array of items. This mode is handy if you want to treat each item independently (though you could also loop over items in the default mode).

Inputs and Outputs: In a Code node, you must explicitly return the data you want to pass to the next node. The return value should be an array of objects (each object representing an item). In JavaScript, you would usually return items; (maybe after modifying them) or construct a new array. For example:

javascript
CopyEdit

```
// Example JS in a Code node (Run Once for All Items)
const results = [];
for (const item of items) {
  const data = item.json;
  // Add a new field or modify data
  data.processedAt = new Date().toISOString();
  data.fullName = data.firstName + " " + data.lastName;
  results.push({ json: data });
}
return results;
```

This code loops through input items, adds two fields (processedAt and fullName), and returns the transformed items. Note how we wrap the JSON object in { json: data } for each result – n8n expects each item in the output array to be an object with a json property (for JSON data). If you have binary data, it would be under a binary property (beyond our scope here).

The Code node provides some helpful global variables and methods:

- $json – shortcut to the current item's JSON (in per-item mode).
- $node – allows you to access data from other nodes by name (e.g. $node["HTTP Request"].json to get the first output item of that node).
- $items("NodeName") – to get all output items of another node.
- There are also utility libraries: in JS, you have access to Luxon (for dates) and JMESPath (for querying JSON), as hinted by the default comments in the node. In Python, you have common libraries like requests for HTTP, etc., enabled by n8n.

Example JSON for a Code node:

json
CopyEdit

```
{
```

```
  "parameters": {
    "language": "JavaScript",
    "mode": "runOnceForAllItems",
    "code": "const items = $input.all();\n// Capitalize a field\nreturn items.map(item => {\n
item.json.name = item.json.name.toUpperCase();\n  return item;\n});"
  },
  "name": "Code",
  "type": "n8n-nodes-base.code",
  "typeVersion": 1,
  "position": [600, 200]
}
```

In this JSON:

- language can be "JavaScript" or "Python".
- mode is set to run once for all (runOnceForAllItems).
- The code field contains the actual script (here we call $input.all() to get all items, then map over them to uppercase a name field).
- The node type is n8n-nodes-base.code. (Older workflows might show type: "n8n-nodes-base.function" with a functionCode parameter – that was the old naming.)

Because the Code node is so free-form, it should be used carefully. Simple transformations (like renaming fields) might be easier with a Set node (described next) or other specialized nodes. But for complex logic, integrations not covered by n8n, or operations across items (e.g., aggregating a sum), the Code node is ideal.

Best Practices for Code Node: If possible, keep the code short and focused. Remember that in JavaScript mode, you can use Node.js built-ins and even require external packages *if* you've enabled that (by default, external modules are not allowed for security, but self-hosted n8n can enable itdocs.n8n.io). Also, handle errors inside the code if needed (you can throw errors which will fail the workflow, or catch them to prevent failure). Use $node to fetch any needed data from earlier in the workflow rather than expecting it in a certain item index. And document your code with comments so others know what it's doing.

IF Node (Conditional)

The IF node introduces basic conditional logic to your workflow. It evaluates a condition (or multiple conditions) on the incoming data and then routes items to either the "true" branch or the "false" branch. Essentially, it asks a yes/no question about each item (or about a set of data) and splits the workflow into two paths.

How it works: The IF node has two output streams: Output *0* for true (condition met) and Output *1* for false (condition not met). You define one or more conditions in its parameters.

Each condition compares a value (which can be a field value or an expression) against another value using an operator (equals, contains, greater than, etc.). All conditions can be combined with AND/OR logic.

Common uses of IF:

- Checking if a number is above/below a threshold (e.g., if price > 100 then do something).

- Checking if a field exists or equals a certain string (e.g., if status == "error" then alert, else continue).

- Filtering items: Only items that satisfy the IF will go to the true branch; others can be handled or discarded via the false branch.

IF node outputs: If an item meets the condition, it will appear as an output item on the true branch (output 0), and that item will not appear on the false branch. Conversely, if an item fails the condition, it goes to false (output 1) and not to true. Downstream, you can connect different nodes to the true vs false outputs to handle each case separately. For example, you might connect an Email node to the true output to send an alert, and just continue normally from the false output.

Configuration: In the editor, the IF node's UI allows selecting "Value 1" (left side of comparison), an operator (like "is equal to", "contains", etc.), and "Value 2" (right side). Each of these can be a static value or an expression. You also specify the data type (string, number, boolean, etc.) so the node can compare properly. Multiple condition lines can be added; you can switch between *AND* (all conditions must be true) or *OR* (any condition true). There's also an option for negating the condition easily (e.g., "If NOT (conditions)").

Example: Suppose you have items each with a field priority. You want to branch if priority is "high". You set the IF node to check: *If priority (as text) equals "high"*. Now connect the IF node's true output to, say, a Slack node that sends a high-priority alert, and connect the false output to a quieter path (or maybe nothing, if you want to drop low priority items at that point). Only the "high" items will go to Slack.

JSON structure for IF node:

json
CopyEdit

```
{
  "parameters": {
    "conditions": {
      "string": [
        {
          "value1": "={{ $json[\"priority\"] }}",
          "operation": "equals",
          "value2": "high"
        }
      ]
```

```
    }
  },
  "name": "IF",
  "type": "n8n-nodes-base.if",
  "position": [800, 200]
}
```

Here we used an expression for value1 to reference the current item's priority field, and we compare to the static string "high". The conditions object can have string, number, boolean, etc., each as arrays of comparisons. Only one type is used depending on what you choose (string in this case).

Switch Node: A related node is the Switch node, which is like a multi-condition if/else if. It lets you define multiple cases (for example, different strings or numbers) and will route items to different outputs based on the case that matches. If IF is not enough (only two branches), Switch can split into many branches. Under the hood, it's like multiple IF checks in one node.

Set Node (Edit Fields)

The Set node (also called *Edit Fields*) is a simple but handy module to manipulate data fields on items. It allows you to add, remove, or rename fields in the JSON data, as well as set static or computed values.

Common uses of the Set node:

- Selecting a subset of fields: If you have items with many properties but only need a few going forward, you can use Set to keep specific fields (there's an option "Keep Only Set" which, if true, will drop all fields except the ones you define).

- Adding new fields: You can define new field names and assign values or expressions to them. For example, add a field "fullName" which is an expression combining first and last name.

- Modifying fields: Although Set doesn't "modify in place" (it more so adds new fields), you can achieve a modification by setting a field to a new value (overwriting it) or by removing a field and adding a new one.

- Removing fields: Simply list fields under the "Remove" section in the node to drop them from the items.

- Renaming fields: There's no explicit rename option in one step, but you can simulate rename by adding a new field with the old field's value (using an expression), then removing the old field.

Set node parameters:

- In the UI, you add any number of fields. Each field has a name and a value. The value can be static (string/number you type) or an expression referencing other data.

- There is a toggle for each field "Use Expression" and a data type selector.

- "Keep Only Set" (boolean): If true, only the fields you explicitly set will be kept; all others are discarded. If false, it will merge your new/updated fields with the existing data.

- "Remove Fields": a list where you can specify field names to delete from the data.

Example: After an HTTP Request, you might get a lot of data but you only need id, name, and created_date. You can use a Set node to keep only those and perhaps format created_date. For instance:

- Add field id -> value {{$json["id"]}} (just carry through),

- Add field name -> value {{$json["full_name"]}} (maybe the original JSON had full_name, but you want it as name in output),

- Add field date -> value {{ new Date($json["created_timestamp"]).toISOString().split("T")[0] }} (example of converting a timestamp to a date string).

- Check "Keep Only Set".

Now the output of the Set node will be items with only id, name, date fields.

Set node JSON example:

json
CopyEdit

```
{
  "parameters": {
    "keepOnlySet": true,
    "values": {
      "string": [
        { "name": "id", "value": "={{ $json[\"id\"] }}" },
        { "name": "name", "value": "={{ $json[\"full_name\"] }}" },
        { "name": "date", "value": "={{ $jmespath($json, \"created_at | to_string(@)\") }}"}
      ]
    },
    "options": {}
```

```
  },
  "name": "Set",
  "type": "n8n-nodes-base.set",
  "position": [1000, 200]
}
```

In this snippet, keepOnlySet: true means only the listed fields remain. We listed three string fields to set. (We used a JMESPath expression for date just as an example of expression usage; simpler would be to use JS in an expression.)

Merge & Flow Control Nodes

For more complex workflows, n8n provides nodes that control the flow of data beyond linear progression:

- Merge Node: The Merge node takes two input streams (two connections into the same node) and combines them into one output. You can configure how the merge happens. Modes include "Append" (simply concatenate the two item lists), "Wait"/"Pass-through" (wait for both inputs – typically used to synchronize parallel branches), "Merge By Key" (merge items based on a matching key field, similar to a database JOIN), or "Merge By Index" (pair the first item of Input A with first item of Input B, etc.). For example, if you branch your flow into two parallel computations and then want to bring the results together, you could use Merge in "Pass-through" mode where one input simply passes its items and the other input is just waited on (ensuring it completes before continuing). Or use "Merge By Index" to attach data from one branch to data from another branch (the resulting item would have combined fields). In JSON, a Merge node is type: "n8n-nodes-base.merge" with parameters specifying mode (e.g., "append", "mergeByKey" etc., plus the key if needed).

- Switch Node: (Mentioned earlier) for multi-branch conditional flows. You set a "Switch value" and different cases. Each case becomes an output. The node will send items to the output whose case matches the value. If no case matches, there's a default output (usually the last one). This is great for routing data into different streams, e.g., based on a type field.

- Split In Batches (Looping): The Loop Over Items / Split In Batches node allows you to process a large list of items in smaller batches, which is useful for rate limiting or iterative processing. You might have 1000 items but want to handle 50 at a time. The node outputs the first batch of items and then, when connected properly in a loop, can output the next batch when triggered again. This node is a bit advanced: you typically connect its output back into its own input or another part of the flow to create a loop (n8n supports looping via this method). This is used to avoid loading too much data at once or to intermix actions (e.g., process 50 records, then wait, then next 50, etc.).

- Wait node: The Wait node pauses the workflow at that point until a condition is met or a certain time has passed. You can configure it to wait for a specific duration (like

sleep for 5 minutes) or until a specific timestamp, or even indefinitely until manually resumed or an external trigger (like via webhook signal). This is useful if you need to delay part of a process (for example, wait a day before sending a reminder email).

- Delay (Workflow): Similar to Wait, but sometimes you might just use Wait for delays. There's also Interval nodes for recurring waits or use Cron for scheduling.

- NoOp (No Operation): This node literally does nothing. It just passes the input to output unchanged. It's rarely needed, but can serve as a placeholder or to intentionally break a connection and then join it (for better visual arrangement) or to label a step. In JSON it's type: "n8n-nodes-base.noOp".

- Error Trigger: Not a regular flow node, but a special trigger we will discuss in error handling – the Error Trigger is used in separate workflows to catch errors from other workflowsdocs.n8n.io.

- Stop and Error: A core node that immediately stops the workflow and marks it as failed, with a custom error message. You can use this within a branch if you detect a condition that should abort the workflow. For example, after an IF node, if something is invalid you might connect to Stop and Error to halt processing and throw an error (which can then be caught by an Error Trigger workflow).

This is not an exhaustive list of core nodes, but these are the main ones you'll use to orchestrate logic.

Other Utility Nodes (Briefly)

- Date & Time: Formats or shifts dates. You can take a date string or timestamp and format it into another format, or calculate relative dates (like +7 days).

- HTML Parser: Takes an HTML string and can extract content via XPath or CSS selectors. Useful if you need to scrape or parse HTML content (perhaps from an HTTP response that returned HTML).

- CSV Parser / Spreadsheet File nodes: Convert CSV to JSON (and vice versa) if you need to handle CSV files.

- Read Binary File / Write Binary File: For reading files from disk or writing to disk (if n8n has filesystem access; typically in self-hosted scenarios). For instance, reading a local CSV to JSON items.

- Move Binary Data: Converts data between binary and JSON on itemscommunity.n8n.iocommunity.n8n.io. For example, after downloading a file via HTTP (which yields binary data in an item), you might use Move Binary Data to convert that binary (maybe a CSV file) into JSON items.

- Execute Command / SSH: Allows running command-line commands on the server or via SSH on a remote hostdocs.n8n.iodocs.n8n.io. This could be used to run a script, or perform server operations that n8n doesn't have a node for.

- Aggregate: Combines multiple items into one (e.g., aggregate values into an array). Opposite of Split Out which can split one item with list field into many items.

- Filter: This node is essentially a simpler IF that just filters out items that don't meet criteria (it has a similar interface).

- Sort: Sort items by a field.

- Limit: Only let a certain number of items pass (or take the first N).

- Crypto: For hashing or encrypting/decrypting data.

- JWT: Create or verify JWTs (JSON Web Tokens).

- LDAP: Query an LDAP directory.

- Webhook (Response): Already covered (Respond to Webhook node for sending response).

- Execution Data: Access metadata about the current execution (like workflow name, execution ID, etc).

- Summarize (AI): An AI utility that can summarize text (if configured with OpenAI credentials) – part of n8n's AI features.

All these nodes have their own parameters, but they follow the pattern of parameters object in JSON export with relevant fields. If you need details on any specific core node not deeply covered here, the n8n documentation has a section for each (for example, Edit Image node can do basic image transforms, FTP node for FTP file transfers, etc.).

Popular Integration Nodes (Apps and Services)

One of n8n's strengths is its large library of integration nodes for third-party services (sometimes called "app nodes"). These allow you to connect to external apps like Google Sheets, Notion, Airtable, email services, databases, CRM systems, etc., without writing code. Integration nodes usually provide a set of **operations** relevant to that service (e.g., "Create Record", "Update Record", "Get All", etc.) and handle the API calls under the hood. You will often need to configure **credentials** for these nodes (OAuth2, API keys, etc.) through n8n's Credentials system.

We will focus on some of the most widely-used integration modules:

Google Sheets Node

Google Sheets integration is very popular for reading or writing spreadsheet data. n8n's Google Sheets node lets you interact with Google Sheets documents through the Google Sheets APIdocs.n8n.io. Typical use cases include adding a new row of data, reading rows for use in a workflow, updating an existing row, or even creating a new spreadsheet.

Configuration & Operations: The Google Sheets node is a bit unique in that it has multiple *resources* and *operations*. When configuring it, you first choose a **Resource** – e.g. "Document" (meaning the spreadsheet file itself) or "Sheet" (meaning a specific sheet/tab within the spreadsheet). Then you choose an **Operation** under that resource. For example:

- **Resource: Document** – Operations might include *Create* (create a new spreadsheet), *Delete* (delete a spreadsheet).

- **Resource: Sheet (Within Document)** – Operations include *Append or Update Row*, *Append Row* (always add new at end)docs.n8n.io, *Get Row(s)* (read data)docs.n8n.io, *Update Row*, *Delete Row/Column*, *Clear Sheet*, *Delete Sheet*, *Create Sheet*docs.n8n.io, etc. Basically anything you'd do with sheet data.

For most cases, you'll use the "Sheet" resource to work with rows. For example, **Append Row** will add a new row of data to a sheet. You need to specify:

- **Spreadsheet ID** (or pick from a dropdown if credentials are set up; n8n can list your Google Drive files).

- **Sheet Name** or ID (the specific tab in the spreadsheet).

- The data for the row. The node UI lets you map each column. If you have a header row, you can reference column names.

The Google Sheets node supports two modes for sending data:

1. As separate fields – you specify each column's value in a separate field in the node (mapping fixed columns).

2. As raw JSON – you pass an object where keys are column names and values are what to insert. This is useful when the input items already have matching field names.

Credentials: To use Google Sheets, you must set up Google API credentials (OAuth2 or service account). Typically, you'll create a Google Sheets OAuth2 credential in n8n and authorize it with your Google account so n8n can access your sheetsdocs.n8n.io. Once that's done, you select that credential in the node (under the Credentials section).

Example: Let's say you want to log form submissions to a Google Sheet. You'd use "Append Row" on resource "Sheet". For each incoming item (each form submission), the node will add a new row. If your sheet has columns Name, Email, Message, you map those to fields from the incoming JSON (e.g., Name -> {{$json["name"]}}, etc.). The node will then produce output items that reflect the newly added rows (or simply an acknowledgment – some operations return the updated data, some just return an ID or success message).

Google Sheets node JSON example (Append Row):

json
CopyEdit

```
{
  "parameters": {
    "resource": "sheet",
    "operation": "append",
    "documentId": "1x2d3...spreadsheet-id...XYZ",
    "sheetName": "FormResponses",
    "options": { "disableAutoMap": false },
    "fields": {
      "columnKey": "name",
      "value": "={{ $json[\"name\"] }}"
    }
    /* ... (in reality, fields would likely be an array for each column, but simplified here) ... */
  },
  "name": "Google Sheets",
  "type": "n8n-nodes-base.googleSheets",
  "position": [1200, 200],
  "credentials": {
    "googleApi": { "id": "goog-cred-id", "name": "Google Sheets OAuth2" }
  }
}
```

This is a simplified representation. The actual fields structure can include multiple columns; n8n might structure it as an array of field definitions or an object mapping. The key parts are documentId (the spreadsheet identifier from its URL) and sheetName. We also see credentials.googleApi referencing the stored credential. The options.disableAutoMap: false

means we're likely manually mapping fields (if true, an "Auto Map" feature would attempt to map matching field names automatically).

Google Sheets integration supports **batch reads** (Get Rows) which will output multiple items (one per row). It can also use cell ranges or filters. For example, "Get Rows" can return all rows or a range like A1:C100. Be mindful of Google API quotas; heavy usage might require delays.

If an operation you need isn't available, you can always fallback to the HTTP node with Google's API (but Google Sheets node covers most standard actions, as indicated by the built-in operations listdocs.n8n.iodocs.n8n.io).

Notion Node

Notion is a popular workspace app (for notes, databases, etc.), and n8n's Notion node allows you to automate tasks with Notion. The Notion node supports a variety of **resources** such as pages, databases, and blocks, each with relevant operationsdocs.n8n.iodocs.n8n.io.

For example:

- **Database**: operations to *Get* a database schema, *Query* a database (search/filter items in a Notion database).
- **Database Page**: operations to *Create* a new page in a database (i.e., add a new entry), *Update* a page, *Get* page properties.
- **Page**: (generic pages outside databases) operations like *Create* a page, *Update* a page, *Archive* (delete) a page.
- **Block**: append content blocks to a page.
- **User**: get user info.

Use cases: You might use Notion node to add new tasks to a Notion database, to update a record when something happens, or to retrieve content for use elsewhere. For example, automatically create a Notion page with details of a new support ticket.

Credentials: Notion uses an internal integration token (API key) that you get by creating a Notion integration and giving it access to pages/databases. In n8n, you'll set up a Notion credential with that tokendocs.n8n.io. The Notion node will then use it to call Notion's API.

Example: Create a Notion Database Page (i.e., add a new entry to a Notion database). You need the Database ID (from the Notion URL or via a prior node). In the Notion node, you'd set Resource: "Database Page" and Operation: "Create"docs.n8n.io. Then you provide the Database ID and the properties for the new page. The node UI will show fields corresponding to your Notion database's properties (like Title, Tags, etc.), where you can fill in values or map from previous nodes. Notion's API is a bit complex about property types

(dates, selects, etc.), but the node abstracts a lot of that. For instance, to set a title property, you just give a plain text and the node wraps it in the required Notion JSON structure.

Notion node JSON example (Create page in database):

json
CopyEdit

```
{
  "parameters": {
    "resource": "databasePage",
    "operation": "create",
    "databaseId": "abcd1234-...-notion-db-id",
    "properties": {
      "Name": {
        "title": [
          { "text": { "content": "={{ $json[\"taskName\"] }}" } }
        ]
      },
      "Status": {
        "select": { "name": "New" }
      },
      "Description": {
        "rich_text": [
          { "text": { "content": "Created via n8n workflow" } }
        ]
      }
      // etc. (Each Notion property requires a specific structure)
    }
  },
  "name": "Notion",
  "type": "n8n-nodes-base.notion",
  "position": [1400, 200],
  "credentials": {
    "notionApi": { "id": "notion-cred-id", "name": "Notion Account" }
  }
}
```

This example shows the complexity: the "Name" property is a Title type in Notion, so the value is provided as an array of text segments. The "Status" is a select property, provided by name. The n8n node UI helps construct this; you wouldn't typically hand-write this JSON. The key point is that resource and operation define what happens, and you supply the necessary IDs and fields.

The Notion node can also *Search* for pages, retrieve blocks (content of a page), and moredocs.n8n.iodocs.n8n.io. If something isn't directly supported (e.g., a new Notion API feature), you could use the HTTP node with the Notion API, but as of now n8n covers the basics like creating and updating database entries.

Remember to share your Notion pages or databases with the integration (bot) you created, otherwise the API will be unauthorized to access them – a common setup hiccup.

Airtable Node

Airtable is a spreadsheet-database hybrid popular for its ease of use. The Airtable node in n8n lets you create, read, update, and delete records in Airtable basesdocs.n8n.iodocs.n8n.io.

Operations: As listed in the docs, Airtable node supports:

- Append (called "Append" or "Create" record – adds a new row to a table)docs.n8n.io.
- Read a record (by ID).
- List records (retrieve multiple, possibly with a filter or view)docs.n8n.io.
- Update a record (by ID).
- Delete a record.

These correspond to Airtable's API. The typical use case is to treat an Airtable table similar to a database table: e.g., when an event happens, add a row to Airtable; or get all rows from Airtable to use in another system.

Configuration: You need the Base ID and Table Name (or ID). Airtable's API requires an API key (or access token). In n8n, you'll set up Airtable credentials by providing your API keydocs.n8n.io. When using the node, you select that credential. Then specify the base (n8n can list your bases once authenticated) and table.

When creating or updating, you provide field values. The node UI will let you input field names and values (like column names and the data, similar to how the Google Sheets node works). For reading or deleting, you need the Record ID – which is Airtable's unique ID for each row (looks like recXXXXXXXX). You might get this ID from a previous List operation or you store it somewhere when you create records. (If you don't have the record ID, you can't update/delete by value directly through the Airtable node – though you could use the List operation with a filter to find records).

Example: Append a new record to an Airtable table "Customers" with fields Name, Email, Signup Date. You configure Airtable node:

- Operation: Append
- Base: e.g., "CRM Database"

- Table: “Customers”

- Fields: Name = {{$json["name"]}}, Email = {{$json["email"]}}, Signup Date = ={{ new Date().toISOString() }} (just as example). The node will output the created record’s data (including its Airtable record ID) in the JSON.

Airtable node JSON example (Append record):

```
json
CopyEdit
{
  "parameters": {
    "operation": "append",
    "baseId": "appABC123XYZ",        // Airtable Base ID
    "table": "Customers",
    "fields": {
      "Name": "={{ $json[\"name\"] }}",
      "Email": "={{ $json[\"email\"] }}",
      "Signup Date": "={{ $json[\"date\"] }}"
    }
  },
  "name": "Airtable",
  "type": "n8n-nodes-base.airtable",
  "position": [1600, 200],
  "credentials": {
    "airtableApi": { "id": "airtable-cred-id", "name": "Airtable Account" }
  }
}
```

This shows baseId (which you get from Airtable, usually starting with app), and table by name. Under fields, we provide an object mapping column names to values (here using expressions from incoming JSON). The credentials reference an API key or personal access token that’s been configured.

One caveat: Airtable API has strict rate limits (e.g., 5 requests per second per base)twilio.com. If you do a lot of Airtable operations in a loop, you might hit those. A best practice is to space out requests (using a Wait node or the built-in “Batch Size” option in the Airtable node if it has one, or simply allow n8n to naturally throttle if possible). If needed, handle the “429 Too Many Requests” errors by catching them (using Continue On Fail and checking error) and waiting.

Email Nodes (SMTP Email)

n8n provides nodes to send emails or to read emails. The primary one for sending is **Send Email** (SMTP). This node lets your workflow send an email via an SMTP server of your choicedocs.n8n.iodocs.n8n.io. There is also an **Email Send (IMAP)** node for reading emails (trigger), which we touched on in triggers.

Send Email node:

- Operation: The Send Email node has two operations: *Send* and *Send and Wait for Response*docs.n8n.io. The normal use is just *Send*, which fires off an email. The "Send and Wait" is a special operation for an approval workflow – it sends an email with some special link or form and then waits for the recipient to respond (this uses n8n's internal features to capture that response). We'll focus on the simple Send operation.

- SMTP Credentials: You must configure an SMTP account in n8n (host, port, user, password, etc.)docs.n8n.io. Many use cases involve Gmail SMTP (which requires an app password or OAuth), or a service like SendGrid, or your own mail server. Once that credential is set and selected in the node, n8n can log in and send emails.

- Parameters: Key fields are:

    - From Email: The sender address. This can often be fixed (like your email) and might be determined by the SMTP credentials (some SMTP servers ignore what you put here and enforce the account's email). You can also put a name like Your Name <>docs.n8n.io.

    - To Email: The recipient address(es). You can separate multiple with commas or use an array input (depending on UI, likely just comma-separated string)docs.n8n.io.

    - Cc/Bcc: (Optional) for carbon copy and blind copy.

    - Subject: The email subject linedocs.n8n.io.

    - Body: The content of the email. There are options for Text or HTML formatdocs.n8n.io, or both. If you choose HTML, you can provide HTML content (and optionally a plain text version).

    - Attachments: You can attach files if you have binary data (the node can take binary file from previous node and send it as attachment).

    - Additional options: e.g., Reply-To address, whether to include n8n attribution footer (by default, n8n might include a small attribution which you can disable via an optiondocs.n8n.io).

Example: Send a notification email to a user:

- From: alerts@mycompany.com
- To: {{$json["userEmail"]}} (coming from earlier node data)
- Subject: Alert: {{$json["issueTitle"]}}
- Body: maybe a combination of static text and data: “Hello, an issue was detected: {{$json["issueDescription"]}}...”

The Send Email node would then send out an email via the configured SMTP (e.g., Gmail or Mailgun, etc.). If successful, its output is typically just an empty item or a success message; you don’t usually need its output, it’s just an action.

Send Email node JSON example:

json
CopyEdit

```
{
  "parameters": {
    "operation": "send",
    "fromEmail": "My App <noreply@myapp.com>",
    "toEmail": "={{ $json[\"userEmail\"] }}",
    "subject": "Alert: ={{ $json[\"issueTitle\"] }}",
    "emailFormat": "text",
    "body": "Hello,\nAn issue was detected:
{{$json[\"issueDescription\"]}}.\nRegards,\nTeam"
  },
  "name": "Send Email",
  "type": "n8n-nodes-base.sendEmail",
  "position": [1800, 200],
  "credentials": {
    "smtp": { "id": "smtp-cred-id", "name": "My SMTP Account" }
  }
}
```

In this JSON:

- operation: "send" (as opposed to "sendWait").
- fromEmail includes a name and address.
- toEmail is set via expression to a value from the data.

- emailFormat: "text" indicates we're sending plain text body (if it were HTML, we might have an html field or something similar).

- body contains the text body (with an expression inside it for issueDescription; note in actual JSON it might need escaping of quotes or use the expression syntax as above).

- The credentials.smtp references an SMTP credential (with server details stored securely).

The Email node is straightforward. Ensure your SMTP credentials are correct and that the service allows third-party sends (for Gmail, use an app password or enable less secure apps, etc.). Secure handling: Do not put SMTP passwords in the workflow JSON; always use the Credentials feature (which encrypts them in n8n's database and only references by name/ID in the workflow export) – this is part of secure best practices.

Other Integrations (Brief Mentions)

Beyond those, n8n has hundreds of nodes for services like Slack, Dropbox, Twitter, MySQL, PostgreSQL, Stripe, GitHub, Trello, etc. Each of those nodes will have their own set of operations but they follow a similar structure:

- Select the resource/entity you want to work with (e.g., for Slack: Channel or Message; for MySQL: just run query or operations).

- Select an operation (e.g., Slack: Post a message, MySQL: Execute Select/Insert, etc.).

- Provide necessary parameters (fields, IDs, etc.).

- Set up credentials of that service type (OAuth tokens, API keys, database connection strings, etc.).

For example, the Slack node allows sending messages to channels or DMs, retrieving messages, etc. The MySQL node allows you to run SQL queries or insert data into a database (you'd provide a query or the data to insert via fields). The HTTP Request node we covered can substitute for any integration that doesn't have a dedicated node – by manually calling the API, as noted in docs: if an operation isn't supported by a specific node, you can use the HTTP node with the service's API, using the same credentialdocs.n8n.io.

To see all available integration nodes, you can browse n8n's nodes list (in editor or docs). They are grouped by categories like "CRM", "Database", "Messaging", etc. Each node's documentation usually lists the supported operations and any particular requirements.

Workflow JSON Structure and Node Definitions

n8n workflows are saved as JSON. Understanding this structure can help in programmatically generating workflows or troubleshooting. When you copy a workflow or export it, you get a JSON object with two main parts:

- an array of **nodes** (each with their config),
- a **connections** object describing how nodes are connected.

Here's a breakdown of a workflow JSON:

Nodes array: Each node is represented as a JSON object with properties:

- `"name"`: A human-friendly name for the node (must be unique within the workflow). E.g., `"HTTP Request"` or `"Set"`.
- `"type"`: The technical identifier of the node type. Core nodes usually have a prefix `n8n-nodes-base.`. For example, the HTTP Request node's type is `"n8n-nodes-base.httpRequest"`, Set is `"n8n-nodes-base.set"`, Google Sheets is `"n8n-nodes-base.googleSheets"`. This tells n8n which node implementation to use.
- `"typeVersion"`: Version number of the node's definition. This increments when a node gets updated in n8n. For instance, HTTP Request might have `typeVersion: 4` in newer n8n[community.n8n.io](community.n8n.io). Usually represented as a number (sometimes as a float like 4.1 which corresponds to internal version).
- `"position"`: An `[x, y]` coordinate for the node's position on the editor canvas. This is for UI layout and doesn't affect execution, but it's part of the JSON.
- `"parameters"`: An object containing all the configuration specific to that node type. This includes things like the URL and method for HTTP nodes, the code for Code nodes, the field mappings for Google Sheets, etc. Each node type defines its parameters in its documentation. For example, for an IF node, `parameters` will have a `conditions` object. For an HTTP node, `parameters` includes `url`, `method`, etc. Any option or field you set in the editor goes into this object.
- `"credentials"` (optional): An object mapping credential type to a credential reference. If the node uses credentials, this will appear. For example, `"credentials": { "googleApi": { "id": "abc123", "name": "Google Sheets OAuth2" } }`. The key (`googleApi` here) depends on the node – it matches the credential type it needs (Google Sheets node expects a Google API credential). The value has an internal ID and the name of the credential you selected (the name is mostly for readability; the id is what n8n uses to link to the actual stored credential which contains secrets).

Note: When sharing workflow JSON publicly, credentials section might be omitted or the id is meaningless to others. One should re-set credentials when importing a workflow.

- Other fields: e.g., "id" for the node itself may be present (especially in newer exports, nodes have an id which is a unique GUID for the node)community.n8n.io. There could be "notes" if you added any notes to the node (n8n allows adding a note/description). Also "disabled": true if the node was disabled.

Connections object: This describes how node outputs connect to inputs of other nodes. The structure is:

json
CopyEdit

```
"connections": {
  "NodeName": {
    "main": [
      [
        { "node": "OtherNodeName", "type": "main", "index": 0 }
      ]
    ]
  },
  ... more nodes ...
}
```

Essentially, for each node that has an output, it lists the connections from that node's outputs. Each node can have multiple output streams (for example, IF has 2 outputs: index 0 and 1). The "main" array corresponds to the main outputs (most nodes only have one output, at index 0; IF would have two subarrays in "main" – one for true branch connections and one for false branch). Each connection entry specifies which node it connects to ("node": "NameOfTargetNode") and which input index of the target ("index": 0 typically, since most nodes have one input). The "type": "main" usually just signifies it's a standard data connection (as opposed to, say, an error or webhook response connection, which n8n also can have, but those are less common).

For example, if you have Node A feeding Node B, and Node A's name is "Set" and Node B's name is "HTTP Request", the connections might contain:

json
CopyEdit

```
"connections": {
  "Set": {
    "main": [
      [ { "node": "HTTP Request", "type": "main", "index": 0 } ]
    ]
  }
}
```

This means Set's main output (index 0 of main outputs) goes into HTTP Request's main input (input index 0).

If a node has multiple outgoing branches (like IF or Switch), there will be an array element for each output index. If an output has multiple connections (fan-out to multiple nodes in parallel), you'd see multiple objects in that inner array.

Workflow-level settings: In the JSON, outside of nodes and connections, there can be some top-level metadata such as name (workflow name), active (if it's active or not on the server), and possibly settings like timezone or tags. For example:

json
CopyEdit

```
"name": "My Workflow",
"active": false,
"settings": { "timezone": "Europe/Berlin" },
"nodes": [ ... ],
"connections": { ... }
```

If a workflow is set to be an "Error Workflow" (for catching errors from others), that is configured in the workflow settings in the UI, not directly obvious in the JSON (except maybe a flag in settings or the presence of an Error Trigger node).

Creating workflows via JSON: You can construct a workflow JSON manually or via code, and then import it in n8n (or even use the n8n API to create it). The structure we described must be followed. For large workflows, it's easier to design in the editor then adjust JSON if needed for minor tweaks.

Example of a simple workflow JSON: Below is a minimal example of a workflow with three nodes connected sequentially: a Webhook -> Set -> HTTP Request, and then the connections.

json
CopyEdit

```
{
  "name": "Example Workflow",
  "nodes": [
    {
      "parameters": { "path": "example", "methods": ["POST"] },
      "name": "Webhook",
      "type": "n8n-nodes-base.webhook",
      "typeVersion": 1,
      "position": [100, 300]
    },
    {
      "parameters": {
        "values": { "string": [{ "name": "receivedAt", "value": "={{ new Date().toISOString() }}"
}] },
        "options": { "keepOnlySet": false }
      },
      "name": "Set",
      "type": "n8n-nodes-base.set",
```

```
      "typeVersion": 1,
      "position": [300, 300]
    },
    {
      "parameters": {
        "method": "POST",
        "url": "https://example.com/api",
        "authentication": "none",
        "jsonParameters": true,
        "options": {}
      },
      "name": "HTTP Request",
      "type": "n8n-nodes-base.httpRequest",
      "typeVersion": 1,
      "position": [500, 300]
    }
  ],
  "connections": {
    "Webhook": {
      "main": [ [ { "node": "Set", "type": "main", "index": 0 } ] ]
    },
    "Set": {
      "main": [ [ { "node": "HTTP Request", "type": "main", "index": 0 } ] ]
    }
  }
}
```

Reading this: The Webhook triggers on POST at path "/example". It connects to Set. The Set node adds a field receivedAt with current timestamp (and since keepOnlySet is false, it will pass through all original data plus this new field). The Set connects to HTTP Request, which will POST to some API (likely sending the combined data as JSON by default since jsonParameters: true indicates it will send the incoming JSON directly).

Note on JSON vs UI: Some fields in JSON can look different than in the UI for brevity or legacy reasons. For instance, a boolean option might appear only if true. Or an enum might be a lowercase string in JSON. Generally, n8n's export is quite readable. If a node's parameters seem confusing, consulting that node's documentation helps as it often describes the fields. Also, the JSON might include default values explicitly or omit them – n8n's import will assume defaults if not present.

Best Practices for Building n8n Workflows

Building workflows can range from simple linear automations to complex, enterprise-level processes. Here are best practices and tips spanning design, error handling, performance, and security:

1. Error Handling and Workflow Monitoring

Use Error Triggers for global error handling: n8n provides an Error Trigger node that can catch failures in other workflowsdocs.n8n.io. This is set up by creating a separate workflow

(often called an "error workflow") that starts with an Error Trigger node. In any workflow's settings, you can designate an Error Workflow to handle its errorsdocs.n8n.io. If that workflow fails (any node errors out and isn't handled), n8n will automatically start the Error Workflow, passing details of the error and the execution. In the error workflow, you can use nodes like Send Email or Slack to notify you of the issue, including which workflow failed and whydocs.n8n.iodocs.n8n.io. This is crucial for production setups – you don't want silent failures. Set up at least a basic error catcher that emails you the error message and workflow name.

Continue on Fail vs Hard Fail: By default, if a node encounters an error (e.g., an HTTP 404, or a script exception in Code node), it stops the workflow immediately. If you want the workflow to continue despite an error (maybe log it and move on), you can enable "Continue On Fail" in that node's settings (usually under Options for the node). When Continue On Fail is true, the node will capture the error as data (outputting an item with the error info) instead of halting the workflow. You can then check the output: many nodes output an object with an "error" property if they failed under this mode. You could follow up with an IF node: if item contains error, handle it (e.g., branch to an alert), else proceed normally. This is useful for non-critical actions or when you intentionally expect occasional failures that you want to handle gracefully.

Use Try/Catch logic with IF: Even without enabling continue-on-fail, you can sometimes structure workflows in a try/catch style. For example, you might duplicate a portion of nodes – one path goes to a node that might fail, and if it fails it triggers an Error workflow or flags something; or you can use the Execute Workflow node (sub-workflow) to attempt an operation and catch its success/failure based on its output.

Testing and Debugging: Use the n8n Editor's execution preview and the debug panel. You can run a workflow manually and inspect each node's output. For complex transformations, use the Debug node or simply attach a temporary Set or Function node to print (console.log) values (in code node, console logs show up in browser console). n8n also has an Execution list where you can examine past runs and where they failed.

Partial Executions: During development, you can right-click a node and choose "Execute node and continue" or "Execute only this node" to test pieces of the workflow. This helps isolate issues. The UI also allows pinning data on a node (so it uses static test data for subsequent runs).

2. Performance Optimization

Avoid unnecessary loops in workflows: n8n processes items efficiently in most cases (it's not extremely fast for thousands of items because it's single-threaded per workflow by default, but it can handle moderate volumes). If you have a very large dataset (hundreds of thousands of items), consider if you can offload the heavy transformation to a database or external script, or use pagination/batching. The Split In Batches node is helpful to break a huge list into chunks to process sequentially, which can avoid high memory usage or hitting API limits.

Use the right node for the job: For example, if you need to combine data from two sources by a key, use the Merge node in "Merge By Key" mode instead of writing a nested loop in a Code node – it will be clearer and possibly more optimized. If you need to filter items, an IF node with only the true path connected can act as a filter (only items meeting condition continue).

Batch API requests: Some integration nodes allow batch operations (e.g., inserting multiple records in one call). Use those if available instead of one call per item. If not available, at least try to use a code node to batch multiple items into one HTTP request if the API supports it (for example, some APIs allow submitting an array of objects in one POST). This reduces overhead and speeds up workflows.

Parallel vs Sequential: n8n largely executes nodes sequentially for each item. If you have two branches (like two outputs of an IF that both continue), those branches run in parallel after the IF. If you need to perform tasks in parallel (e.g., call two different APIs at the same time) and then wait for both to finish, you can branch and then use a Merge node (in wait mode) to join. But note, heavy parallelism might strain your server or hit rate limits. n8n doesn't spawn new threads for parallel branches; it interleaves them in the single event loop.

Resource considerations: If self-hosting n8n, be mindful of memory. For example, reading a huge CSV (100k rows) into memory (as 100k items) might be heavy. Instead, if possible, stream it or use the built-in Streaming capabilities (n8n can stream through nodes like the HTTP node for large binary but not all nodes support streaming processing of items). For extremely large data, consider connecting a database and processing in SQL.

Queue Mode and Scaling: n8n has a "queue mode" where each execution can run in a separate process (backed by a Redis queue). This is more advanced, but if you find one workflow is blocking others or consuming all CPU, queue mode could be enabled to distribute load. That's more for enterprise scaling – for normal use, a single process handling a few workflows concurrently is fine.

3. Workflow Design and Maintainability

Keep workflows modular: If your workflow is getting very large or does multiple distinct things, consider breaking it into sub-workflows. n8n allows one workflow to call another using the Execute Workflow (Sub-workflow) nodedocs.n8n.iodocs.n8n.io. This can greatly improve clarity and reuse. For instance, if several workflows need to send a formatted report via email, you could create one sub-workflow that generates the report and sends the email, and call it from different places. Or if a workflow has a repeating sequence (e.g., process each record in a list in the same way), it might be cleaner to run that sequence as a sub-workflow for each item, rather than copying the nodes or complicating with loop logic.

Benefits of sub-workflows: Reusability, and also if a sub-workflow fails or needs changes, it's isolated. According to n8n's docs, sub-workflows are useful for breaking large workflows into smaller components and reusing themdocs.n8n.io. To implement, you create a workflow with an Execute Workflow Trigger (called "When Executed by Another Workflow" in the editor) as its start node. Then in the main workflow, use Execute Sub-workflow node to call it, passing along data. The sub-workflow can return data back to the main flow (it outputs whatever the last node in it outputs, back into the Execute Workflow node's output).

Version control & documentation: Use n8n's export feature to save JSON of your workflows in a Git repository if you want version control. Add descriptions to workflows and nodes (there's a field for workflow description and you can add sticky notes or node notes) to document the purpose of each part. This is extremely helpful when you or others revisit the workflow later.

Use naming conventions: Name your nodes descriptively. Instead of leaving it as "Function1" or "Set", rename it to "Calculate Total Price" or "Format Date" etc. This makes the visual flow easier to follow. Similarly, keep your workflows named clearly with their purpose.

Testing sub-parts: If your workflow has multiple paths, test each path. You can temporarily inject test data using the Inject or Manual triggers or by using the Function node to generate sample data (there's actually a core node called Manual Trigger you can use to manually start with some dummy data, or just start with a Set node that defines some sample fields for testing).

4. Secure Credential Handling and Data Security

Never hard-code secrets in workflows: n8n's credential system is designed to store API keys, passwords, etc., securely (usually encrypted in the database) and only reference them by name in the workflow. Always use Credentials for things like database passwords, API tokens, SMTP logins, etc. For example, in the HTTP Request node, rather than putting an API key in the headers manually each time, create an HTTP Header Auth credential or a dedicated credential type if available. This way, if the key changes you update it in one place, and exports of the workflow don't contain the secret. The workflow JSON will only show a referencecommunity.n8n.io, like:

json
CopyEdit

```
"credentials": {
  "httpHeaderAuth": {
    "id": "ASbkjh312",
    "name": "My API Credential"
  }
}
```

and not the actual key.

Use environment variables for credentials if possible: n8n (when self-hosted) can be configured to load credential data from environment variables. This allows secrets to reside in Docker env vars or server config rather than even in n8n's database. If using n8n Cloud, the credentials are managed in their system, but still safe.

Limit sensitive data exposure: If your workflow deals with personal data or other sensitive info, be mindful when using nodes like Webhook or external calls that might log data. Avoid storing sensitive data unencrypted in data stores – e.g., if you are moving data through n8n just to send somewhere secure, try not to keep it in plain text in an execution log. You can turn off saving successful executions or specify data to remove after execution if needed in workflow settings (there's a setting to not save execution data, or you can configure n8n to save only error executions in production). This reduces risk of sensitive data lingering.

Error workflow security: If you use an Error Trigger workflow to email you errors, be cautious not to include overly sensitive data in that email (the error trigger provides error message and stack trace, which might include pieces of items). It might be enough to send the workflow name and error summary, and then you can log into n8n to inspect details.

Sub-workflow trust: The Execute Workflow node allows specifying which workflows can be called (for security, to prevent unauthorized calling of workflows). If you have confidential sub-workflows, you can restrict them to only be callable by certain parent workflows in the Workflow Settings ("This workflow can be called by…" setting)docs.n8n.io. This is more relevant in multi-user scenarios.

Regularly update n8n: This is a general best practice – ensure you're using an updated version of n8n, as updates often fix bugs and sometimes patch security issues or improve nodes.

Handling credentials in Code node: n8n does not directly expose credential values inside the Code node (for good reason). If you need to use a credential in custom code, one workaround is to use the HTTP Request node (with that credential) or another appropriate node and then use code to process results. There is also a notion of "External secrets" and environment variables: you can use expressions like {{$env("MY_SECRET")}} in nodes to inject environment variable values (set in your server environment). That can be used in a Code node as well via an expression. This is advanced, but know that plain text secrets in Code is not recommended.

5. Workflow Efficiency and Organization

Use Comments (Sticky Notes): n8n has a "Sticky Notes" feature where you can add colored note boxes on the canvas. Use these to explain sections of the workflow, especially if it's complex. For example, a note that says "Loop through API results pages" around nodes handling pagination, or "This branch handles high priority cases". It greatly aids anyone reading the workflow (including future you).

Avoid overly large single workflows if possible: There's no hard limit, but very large workflows can become difficult to manage in the editor and to debug. If you find a workflow has many branches and sub-branches, see if logically some parts can be separate. Perhaps a trigger workflow receives data and then fires off different other workflows (maybe via Webhook calls or the built-in "Execute Workflow").

Testing on copies: When modifying a critical workflow, it can be wise to clone it (n8n allows duplicating a workflow) and test changes on the clone with sample data. Once you're confident, either swap them or apply changes to the original and activate it. This avoids breaking a running workflow while you iterate on changes.

Use Source Control for Logic (if complex): If your workflow uses custom code (Code nodes), treat that code as you would any script – test it, and perhaps keep a copy in a Git repository or at least in a file for versioning. While the rest of the workflow is configuration, those code blocks are logic that might need debugging. Keep them as small as possible, and consider writing unit-like tests for those code segments outside n8n if they're complex (i.e., write a separate JS function and test it, then embed it in the Code node).

Real-World Workflow Scenarios and JSON Examples

To solidify understanding, let's walk through a couple of real-world scenarios that combine multiple nodes. We will outline each scenario, explain which nodes are used and why, and provide simplified JSON snippets illustrating how the workflow is structured.

Scenario 1: Webhook Form to Notion and Email Notification

Use Case: Imagine you have an online form (on a website) where users submit feedback. You want n8n to handle each submission by creating a new page in a Notion database (to log the feedback) and then send a thank-you email to the user. This scenario involves:

- a Webhook trigger to catch form submissions (with data like name, email, message),
- a Notion node to insert the data into a Notion table (or database page),
- a Send Email node to email the user a thanks.

Workflow Outline:

1. Webhook (Trigger) – Starts the workflow when the form POSTs data to it. The incoming data includes fields: name, email, message.
2. Notion (Create Database Page) – Uses the Notion node to create a new entry in the "Feedback" database. Maps the form fields into Notion properties (e.g., Name, Email, Message, Date).
3. Send Email (SMTP) – Sends an email to the submitter's email address. The email content thanks them for their feedback.

We might also include a Set node or Function node in between to format data (for example, maybe combine name fields or format date). But let's assume the form gives data in the correct format.

Notion Setup: We have a Notion database "Feedback" with properties: *Name* (text), *Email* (text), *Message* (text), *SubmittedAt* (date). We will fill those.

Email Setup: We have SMTP credentials configured (e.g., Gmail) and will use that.

Flow of data: Webhook receives JSON like:

json
CopyEdit
{ "name": "Alice", "email": "alice@example.com", "message": "Great product!" }

This becomes item.json for the next nodes.

Notion Node configuration: Resource = Database Page, Operation = Create. Database ID is pre-set (from Notion). We map:

- Name -> {{$json["name"]}}
- Email -> {{$json["email"]}}
- Message -> {{$json["message"]}}
- SubmittedAt -> current date (we can use an expression like {{ new Date().toISOString().split('T')[0] }} for date-only, or leave blank to let Notion auto-set if configured).

Email Node configuration:

- To = {{$json["email"]}} (from webhook data, still available in flow)
- Subject = “Thanks for your feedback, {{$json["name"]}}!”
- Body = “Hi {{$json["name"]}},\n\nThank you for your message: "{{$json["message"]}}".\nWe appreciate your feedback!\n\nBest regards,\nTeam”

We will assume these nodes are connected linearly: Webhook -> Notion -> Email. (Alternatively, we could do Notion and Email in parallel by branching from the Webhook, but sequential is fine here).

Example JSON for this workflow:

json
CopyEdit

```
{
  "name": "Feedback to Notion and Email",
  "nodes": [
    {
      "parameters": {
        "path": "feedback",
        "methods": ["POST"]
      },
      "name": "Webhook",
      "type": "n8n-nodes-base.webhook",
      "position": [100, 300]
    },
    {
      "parameters": {
        "resource": "databasePage",
        "operation": "create",
        "databaseId": "abcd-efgh-...NotionDBID...",
```

```
        "properties": {
          "Name": {
            "title": [{ "text": { "content": "={{ $json[\"name\"] }}" } }]
          },
          "Email": {
            "email": "={{ $json[\"email\"] }}"
          },
          "Message": {
            "rich_text": [{ "text": { "content": "={{ $json[\"message\"] }}" } }]
          },
          "SubmittedAt": {
            "date": { "start": "={{ new Date().toISOString() }}" }
          }
        }
      },
      "name": "Notion",
      "type": "n8n-nodes-base.notion",
      "position": [300, 300],
      "credentials": {
        "notionApi": { "id": "notion_cred_id", "name": "Notion API Token" }
      }
    },
    {
      "parameters": {
        "operation": "send",
        "fromEmail": "Support Team <support@mycompany.com>",
        "toEmail": "={{ $json[\"email\"] }}",
        "subject": "Thank you for your feedback, {{ $json[\"name\"] }}",
        "emailFormat": "text",
        "body": "Hi {{ $json[\"name\"] }},\n\nThank you for your message: \"{{
$json[\"message\"] }}\".\nWe appreciate your feedback!\n\nBest regards,\nThe Team"
      },
      "name": "Send Email",
      "type": "n8n-nodes-base.sendEmail",
      "position": [500, 300],
      "credentials": {
        "smtp": { "id": "smtp_cred_id", "name": "SMTP Account" }
      }
    }
  ],
  "connections": {
    "Webhook": {
      "main": [ [ { "node": "Notion", "type": "main", "index": 0 } ] ]
    },
    "Notion": {
      "main": [ [ { "node": "Send Email", "type": "main", "index": 0 } ] ]
    }
  }
}
```

Explanation: The Webhook triggers on POST /webhook/feedback (n8n's URL will include workflow ID and path "feedback"). Its output (the form JSON) flows into the Notion node. The Notion node uses that data to create a page in the Notion database (the JSON shows how Notion properties are structured in the API; the node handles this complexity, but in JSON export we see it). Then the Notion node passes the same input to the Email node (by default, if Notion successfully created the page, it outputs the created page data. But here we might actually want the original form data still. One trick: the Notion node output will not contain the original email because Notion's response won't echo that directly. We might need to adjust: either use an IF or Merge to bring the original data to the email node. Alternatively, we could configure the Notion node's Return Data option to include the input data. For simplicity, assume we had a Merge node or we set Notion to passthrough original fields). The Email node then sends to the user's email with subject and body referencing their name and message.

This workflow demonstrates connecting multiple services: a custom webhook, a SaaS app (Notion), and email.

Scenario 2: Scheduled API Data Fetch and Google Sheets Update

Use Case: You want to run a daily job that fetches data from a public API and appends the results to a Google Sheets spreadsheet, then sends an email summary if certain conditions are met. For example, every day at 9:00 AM, call a weather API to get today's forecast. Add a new row in a Google Sheet logging the date and forecast. If the forecast indicates rain, send an email alert to bring an umbrella.

Workflow Outline:

1. Schedule Trigger – Cron set to 9:00 AM daily.

2. HTTP Request – GET request to the Weather API (e.g., OpenWeatherMap or similar) to get today's forecast for a city.

3. IF – Check the response for "rain" in the forecast.

    - True branch: will trigger an alert email.

    - False branch: do nothing (or could go to next step directly).

4. Google Sheets – Append Row to log data in a spreadsheet (e.g., columns: Date, Forecast, Temp).

5. Send Email (on the true branch of IF) – email yourself "It will rain today - don't forget umbrella."

Data flow considerations: The HTTP Request returns JSON with forecast details. We may need a Function or Set node to extract the parts we want to log. Let's say the API returns JSON like:

json
CopyEdit
{ "date": "2025-04-22", "weather": "Rainy", "temp": 18, "description": "Light rain in the morning." }

(This is hypothetical for simplicity). We want to append to Google Sheets: Date = date, Weather = weather, Temp = temp, Description = description.

We will assume the HTTP node outputs something similar to above (if not, we might use a Code node to map the API's actual response fields to these names).

IF Node: Condition: weather contains "Rain" (or equals "Rainy"). If true (forecast has rain), go to Email.

Connections:

- Schedule -> HTTP Request -> (both IF and Google Sheets). Actually, here we have a decision: we want to log to Google Sheets regardless of rain or not. And we want to possibly send email if rain. We can design it two ways:

    - Sequence: HTTP -> IF. On IF's true branch, send Email then (optionally) also connect to Google Sheets; on false branch, directly to Google Sheets. This ensures we always log to sheet, and conditionally email.

    - Or do HTTP -> Google Sheets (always log), then also an IF branching off from HTTP output in parallel to handle email. However, if we do parallel, we must ensure the HTTP output is available to both branches. n8n can split outputs by connecting the HTTP node to two nodes (one IF, one Sheets) – it will duplicate the data to both connections (this is a feature: one node's output can feed multiple next nodes).

We'll do parallel branches from HTTP: one branch goes to Google Sheets append (no condition, always executes), another branch goes to IF -> Email for the alert. This parallel design means the order isn't strictly guaranteed, but that's fine. Or we could ensure sequence (log then email or email then log). It doesn't matter much here, but logging even if the email fails might be good.

Credentials: We need an API (if the weather API requires a key, either we use HTTP node with an API key param or credentials if it had a special auth). We also have Google Sheets credential (OAuth to Google account), and SMTP for email (or could send to ourselves via Gmail API as well, but we'll just use SMTP as before).

Workflow JSON (simplified):

json
CopyEdit

```
{
  "name": "Daily Weather Log and Alert",
  "nodes": [
    {
      "parameters": {
        "mode": "everyDay",
        "hour": 9,
        "minute": 0
      },
      "name": "Schedule Trigger",
      "type": "n8n-nodes-base.scheduleTrigger",
      "position": [100, 250]
    },
    {
      "parameters": {
        "method": "GET",
        "url":
"https://api.weather.example.com/today?city=London&units=metric&apiKey=XYZ",
        "responseFormat": "json"
      },
      "name": "HTTP Request",
      "type": "n8n-nodes-base.httpRequest",
      "position": [300, 250]
      // Assume no special auth needed for simplicity (or apiKey in URL as query).
    },
    {
      "parameters": {
        "conditions": {
          "string": [
            {
              "value1": "={{ $json[\"weather\"] }}",
              "operation": "contains",
              "value2": "Rain"
            }
          ]
        }
      },
      "name": "IF Rain?",
      "type": "n8n-nodes-base.if",
      "position": [500, 150]
    },
    {
      "parameters": {
        "resource": "sheet",
        "operation": "append",
        "documentId": "1AbCdEfGhIjKlMnOpQrStUvWxYz",  // Google Sheet ID
        "sheetName": "DailyForecast",
        "fields": {
          "Date": "={{ $json[\"date\"] }}",
          "Weather": "={{ $json[\"weather\"] }}",
```

```
        "Temp": "={{ $json[\"temp\"] }}",
        "Description": "={{ $json[\"description\"] }}"
      }
    },
    "name": "Google Sheets",
    "type": "n8n-nodes-base.googleSheets",
    "position": [500, 350],
    "credentials": {
      "googleApi": { "id": "google_sheet_cred_id", "name": "Google Sheets OAuth2" }
    }
  },
  {
    "parameters": {
      "operation": "send",
      "fromEmail": "Weather Bot <weatherbot@mydomain.com>",
      "toEmail": "me@myemail.com",
      "subject": "Rain Alert: Forecast says Rain today",
      "emailFormat": "text",
      "body": "Don't forget your umbrella! Today's forecast: {{ $json[\"description\"] }}."
    },
    "name": "Send Email",
    "type": "n8n-nodes-base.sendEmail",
    "position": [700, 150],
    "credentials": {
      "smtp": { "id": "smtp_cred_id", "name": "SMTP Account" }
    }
  }
 ],
 "connections": {
  "Schedule Trigger": {
    "main": [ [ { "node": "HTTP Request", "type": "main", "index": 0 } ] ]
  },
  "HTTP Request": {
    "main": [
      [ { "node": "IF Rain?", "type": "main", "index": 0 } ],
      [ { "node": "Google Sheets", "type": "main", "index": 0 } ]
    ]
  },
  "IF Rain?": {
    "main": [
      [ { "node": "Send Email", "type": "main", "index": 0 } ],
      []
    ]
  }
 }
}
```

Explanation: The Schedule trigger kicks off daily. The HTTP Request calls the weather API. In connections, you see HTTP Request has two outputs: one going to IF, another to Google Sheets. Actually, HTTP Request node has only one output (index 0), but we have connected it to two nodes, so in JSON it appears as two entries in the connections list for HTTP

Request's main output. n8n will feed the same items to both next nodes in parallel[docs.n8n.io](docs.n8n.io) (meaning the weather JSON goes to the IF node and to the Google Sheets node independently). The IF checks if the weather string contains "Rain". If true, it goes into output 0 which is connected to Send Email (if false, nothing happens on output 1 – we left that empty array). Regardless of rain, the Google Sheets node will execute and append the row (because it's not gated by the IF in this design).

We log Date, Weather, Temp, Description. We assumed the HTTP response had those fields at top-level; if not, we might need a little transformation. (Alternatively, we could use the JMESPath or just expressions to pick fields from a nested JSON; e.g., if API gave forecast.description, we'd use {{$json["forecast"]["description"]}} in the Google Sheets fields.)

After appending to Google Sheets, we might want to know if it succeeded or handle errors, but in this simple case we assume success (if it fails, the workflow stops and Error Trigger could catch it).

The email is only sent if the IF condition matched, with a simple body advising about umbrella and including the forecast description (like "Light rain in the morning."). The {{$json["description"]}} in the Email node refers to the data from the HTTP node (because that's what is flowing through the IF true branch – by default the IF node passes the same item through if it meets condition). So $json["description"] is available there.

This scenario shows a scheduled integration, conditional branching, and parallel node execution, as well as logging to a sheet and conditional notification.

Scenario 3: Using Sub-Workflow for Reusability

Use Case: Suppose you have multiple workflows that all need to perform a common task – for example, sanitizing and validating an email address (removing whitespace, converting to lowercase, checking it matches a pattern). Instead of duplicating the same nodes in each workflow, you can create a sub-workflow that does this and call it from the main workflows.

Sub-workflow design:

- Workflow B: "Email Sanitizer" – starts with Execute Workflow Trigger (so it can be called), then perhaps a Function node that takes an input email (from $json["email"]) and outputs a sanitized version or flags invalid. It could output something like { "email": "...", "valid": true/false }.

- Main Workflow A: collects some data, has an email field that needs cleaning, uses Execute Workflow (Sub-workflow) node to call Workflow B, passes the email, receives the cleaned result, then continues (maybe only continues if valid is true, else uses Stop and Error or something).

For demonstration, we won't show the entire main workflow, just how the sub-workflow is invoked.

Execute Sub-workflow Node (in main) configuration:

- Select Workflow: “Email Sanitizer”.

- Input: There are options – either you map specific fields (if the sub-workflow’s trigger node defined expected fields) or you just pass all data. Let’s assume we only need to pass the email. n8n’s Execute Workflow can automatically pass the current item’s JSON to the sub-workflow. Our sub-workflow expects email in the input item JSON.

Sub-workflow JSON (simplified):

json
CopyEdit

```
{
  "name": "Email Sanitizer",
  "nodes": [
    {
      "parameters": {},
      "name": "Execute Workflow Trigger",
      "type": "n8n-nodes-base.executeWorkflowTrigger",
      "position": [100, 200]
    },
    {
      "parameters": {
        "language": "JavaScript",
        "code": "let email = $json[\"email\"] || \"\";\nemail =
email.trim().toLowerCase();\nconst valid = /\\S+@\\S+\\.\\S+/.test(email);\nreturn [{ json: {
email, valid } }];"
      },
      "name": "Clean Email",
      "type": "n8n-nodes-base.code",
      "position": [300, 200]
    }
  ],
  "connections": {
    "Execute Workflow Trigger": {
      "main": [ [ { "node": "Clean Email", "type": "main", "index": 0 } ] ]
    }
  }
}
```

This sub-workflow simply takes the incoming data (which should have an email field), runs a short JS code to trim whitespace and lowercase it, then uses a regex to validate format (very simple validation), and returns an item with email and valid. We set up Execute Workflow Trigger as the start (no special parameters needed except maybe defining input schema, but leaving it open “Accept all data” modedocs.n8n.iodocs.n8n.io). The Code node returns the processed data.

Main Workflow snippet where it's called:

json

CopyEdit

```
{
  "parameters": {
    "workflowId": "Email Sanitizer"
    // or use workflow name or ID depending on config
  },
  "name": "Execute Email Sanitizer",
  "type": "n8n-nodes-base.executeWorkflow",
  "position": [400, 200]
}
```

Connections would have the previous node connecting into this, and then this connecting to something after.

When main workflow runs, at that node, it will call the sub-workflow. The sub-workflow runs its nodes and returns output. The "Execute Workflow" node then outputs whatever the sub returned. In our case, it would output the cleaned email and validity flag. The main workflow can then use an IF node to check valid == true and proceed, or handle invalid.

This scenario showcases how to break a problem into a re-usable component using sub-workflows – improving maintainability (if you change the cleaning logic, you do it in one place).

These scenarios illustrate how n8n modules (nodes) work together in practice:

- A trigger gathers input (webhook or schedule).
- Data is passed through transformation or logic nodes (IF, Code, Set).
- Integration nodes interface with external systems (Notion, Google Sheets, HTTP API).
- Actions like sending email provide outputs to the world.
- Branching logic (IF) and parallel execution allow complex conditional flows.
- Sub-workflows enable modular design, so repetitive tasks can be abstracted.

By understanding each node's role and the JSON structure that defines it, an AI Agent or developer can reason about constructing workflows for a given task. The key is to break the task into steps, choose appropriate nodes for each step (use core nodes for logic and transformation, use integration nodes for external services, handle errors appropriately), and connect them in the right order. Always test with sample data and iterate.

Conclusion: This guide has covered all major n8n modules/nodes – from triggers (Webhook, Schedule) and core logic nodes (HTTP Request, Code, IF, Set, etc.) to popular integration nodes (Google Sheets, Notion, Airtable, Email). We also delved into how to represent and assemble these nodes in JSON, and best practices to build robust, maintainable workflows. With this knowledge base, you should be well-equipped to design n8n workflows that connect tools and automate tasks effectively, whether manually or by instructing an AI agent to do so. Happy automating!

Sources:

- n8n Documentation – *Understanding the data structure of n8n*docs.n8n.io
- n8n Documentation – *HTTP Request node* (overview)docs.n8n.io
- n8n Documentation – *Using the Code node* (Function/Code node info)docs.n8n.iodocs.n8n.io
- n8n Documentation – *Webhook node* (trigger usage)docs.n8n.io
- n8n Documentation – *Google Sheets node operations*docs.n8n.iodocs.n8n.io
- n8n Documentation – *Notion node* (features and ops)docs.n8n.iodocs.n8n.io
- n8n Documentation – *Airtable node* (operations)docs.n8n.iodocs.n8n.io
- n8n Documentation – *Send Email node*docs.n8n.iodocs.n8n.io
- n8n Documentation – *Dealing with errors in workflows* (Error Trigger usage)docs.n8n.io
- n8n Documentation – *Execute Sub-workflow Trigger node* (sub-workflow usage)docs.n8n.iodocs.n8n.io