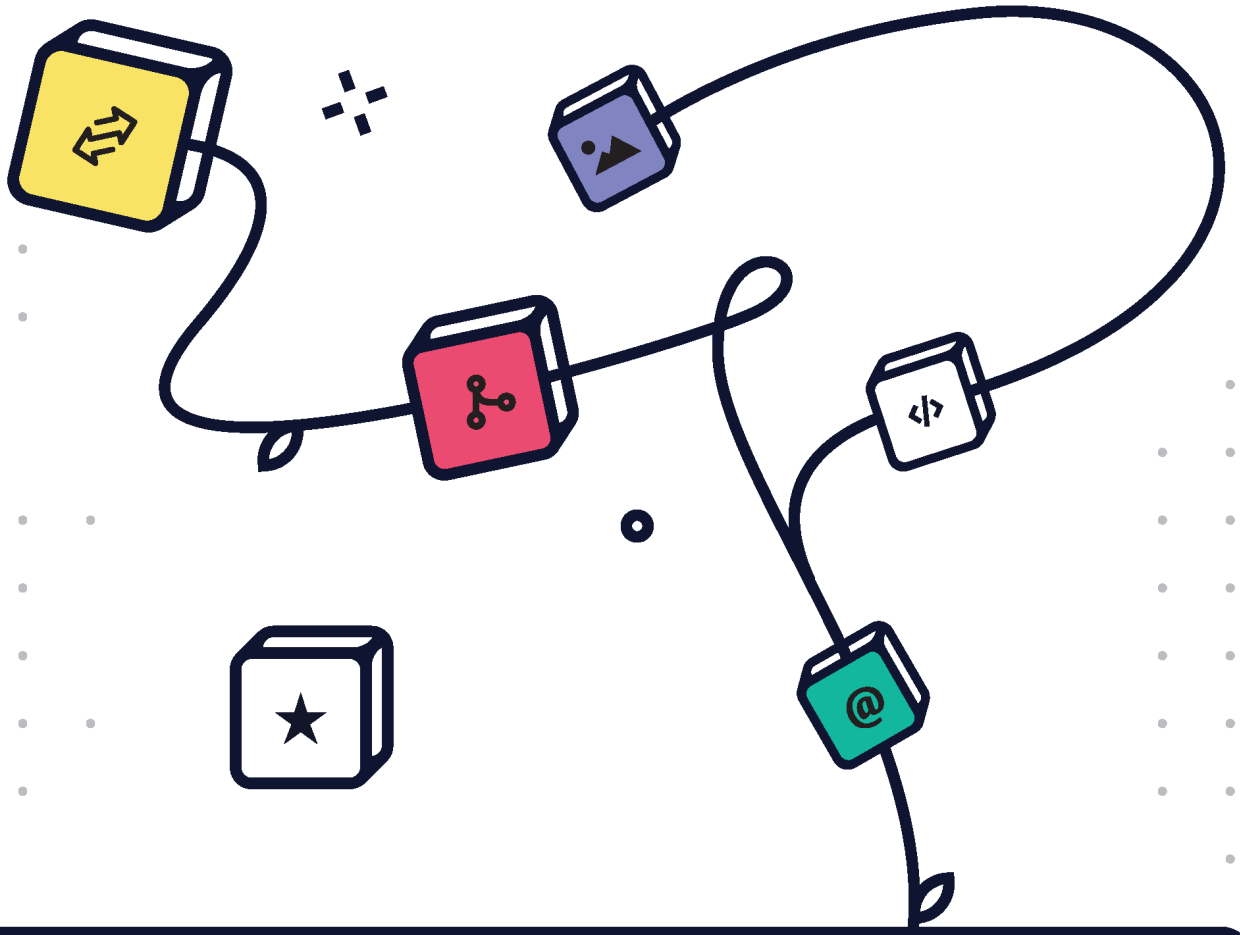

Rapid Product Development with n8n

Practical guide to creating digital products on the web using workflow automation and n8n

Jason McFeetors | Tanay Pant

n8n

Rapid Product Development with n8n

Practical guide to creating digital products on the web using workflow automation and n8n

Jason McFeetors

Tanay Pant

n8n IS AN IMPRINT OF PACKT PUBLISHING

Rapid Product Development with n8n

Associate Group Product Manager: Pavan Ramchandani
Publishing Product Manager: Dhruv Kataria
Senior Editor: Sofi Rogers
Content Development Editor: Feza Shaikh
Technical Editor: Shubham Sharma
Copy Editor: Safis Editing
Project Coordinator: Manthan Patel
Proofreader: Safis Editing
Indexer: Pratik Shirodkar
Production Designer: Ponraj Dhandapani
Marketing Coordinator: Teny Thomas

First published: February 2022

Production reference: 1230222

Published by Packt Publishing Ltd.
Livery Place
35 Livery Street
Birmingham
B3 2PB, UK.

978-1-80181-736-3

www.packt.com

To my wife, Janelle, and daughters, Rayna and Caris, who put up with endless conversations about writing, technical documentation, JSON, and APIs. I would not have been able to do this without them as my inspiration.

– Jason McFeetors

To my companion and best friend, Isabel, who helped me fight procrastination and finish this book in time. To my family and everyone who has supported me with all the things I have been working on. It's easier standing on the shoulders of giants :)

– Tanay Pant

Foreword

I created n8n, an open and free (`https://faircode.io/`) workflow automation tool, in 2019. Since its inception, n8n has attracted creators and developers to build digital products online quickly. To fuel the learning and utility of n8n, Tanay and Jason have come up with this step-by-step guide to n8n that will get you up and running with it in no time.

Tanay and Jason are tech advocates who work closely with the community. They have distilled their years of experience in this hands-on book that will make you appreciate the versatility of n8n. This book enables all web developers to build apps that automate repetitive tasks and make things easier for teams and start-ups. I am fascinated by the easy-to-follow approach of this book about n8n. We created n8n to be an easy-to-use platform. However, with its range of capabilities, a good guide is a must for everyone taking their first step in exploring n8n. This book does that job for everyone.

I recommend this book and encourage people to take the journey into the world of n8n. Finally, I hope you enjoy reading the book as much as I did.

Jan Oberhauser

Founder and CEO, n8n.io

Contributors

About the authors

Jason McFeetors works as a senior management consultant for a large IT consulting firm in Canada, where he is presently working with a team designing and building cloud-based call center solutions. Jason has been working in the IT industry for over 25 years and has worked at nearly every level in the industry, from support tech to system architect to chief technology officer. His love for all things tech takes him to all sectors of the industry, including software development, hardware design, automation, and IoT. His work has previously been featured in *Popular Science* and *Lifehacker*.

If it had not been for Tanay going out on a limb and asking me to work on this project with him, it would have never crossed my mind. You are a true inspiration. And to the n8n team and community, your support and encouragement have allowed me to believe the impossible is within my grasp.

Tanay Pant is an author, speaker, and developer relations expert. He has written the books *Learning Web-based Virtual Reality, Building a Virtual Assistant for Raspberry Pi*, and *Learning Firefox OS Application Development*. He has been listed in the *about* credits of the Firefox web browser for his contributions to the different open source projects of the Mozilla Foundation. He also writes for several websites, such as SitePoint and Tuts+, where he shares tips and tricks about web development.

I would like to thank my co-author, Jason, and the whole n8n team for supporting me with this book and for nurturing the next generation of builders.

About the reviewers

Max Tkacz is a principal product designer. He's a low-code evangelist who's focused on building and scaling products that empower makers to create more easily. He is experienced in automation, blockchain, and fintech. He works to give automation superpowers to all as head of design at n8n.

Omar Ajoue was born and raised in Brazil, where he studied computer science.

His life in technology started early, at the age of 10, when he would dismantle broken toys to understand how they worked. Piles of batteries, wires, motors, and electronic components were part of his collection.

He always loved tinkering, but mostly destroying, in order to learn. He was never focused on building stuff until he got his first job as a programmer.

With no previous experience, he spent 7 years working for the same company, which taught him his future craft: software development.

With broad experience in the IT industry, Omar worked as a software engineer, team lead, and CEO and is now a senior software engineer for n8n.

Table of Contents

3

Diving into Core Nodes and Data in n8n

4

Learn by Doing: Building Two n8n Apps

Section 2 – Building an API to Power Your Application

5

Building Your First API Endpoints

6

Powering Your API with a No Code Database

Section 3 – Building the User Interface and Connecting the API

9

Building the User Interface of the Application

10

We've Only Just Begun

Preface

n8n enables users to connect different systems and cloud services without needing expensive developers or technical skillsets. It allows you to reduce the time required to develop new products, helping you bring them to the market quicker than if you had to muster a whole development team.

Developers working with n8n will be able to put their knowledge to work with this practical guide to building low-code applications. The book takes a hands-on approach to implementation and associated methodologies that will have you up and running and productive in no time.

You will begin by learning about where n8n fits in the tech stack of your business and how it provides opportunities for reducing costs as well as increasing efficiency and revenue. Later, you will identify opportunities where you can leverage n8n's connectivity and automation functionality within your working environment and progress to building out an n8n-based toolset that will immediately have a positive effect on your operation's bottom line.

By the end of this book, you will be able to identify real-world opportunities to generate income, improve efficiency, and build tools to capitalize on those opportunities.

Who this book is for

This book is for web developers and low-code enthusiasts who have basic knowledge of the JavaScript programming language and web concepts such as APIs and webhooks. The book assumes beginner-level knowledge of JavaScript programming.

What this book covers

Chapter 1, Introduction to No Code, n8n, and Bubble, is where you will gain an understanding of no code and why it is becoming important to business and technology. You will also become familiar with n8n, a no code automation tool, and Bubble, a no-code development platform.

Chapter 2, Using the Editor UI to Create Workflows, teaches you how to use n8n's Editor UI. You will also learn about the different kinds of nodes in n8n and how to use them. You will then learn about workflows, deal with dynamic data, and finally create your first workflow in n8n.

Chapter 3, Diving into Core Nodes and Data in n8n, is where you will learn how to use n8n's data structure to manipulate and transform data inside workflows, and use JavaScript inside your low-code workflows to unlock custom functionalities. You'll call REST APIs from inside your workflows using the HTTP Request node. The chapter will also cover how to handle real-time events using the Webhook node and how to trigger workflows based on this data. You'll work with arrays and JSON objects inside n8n, understand when to use what kind of data format, and transform data inside your workflows.

Chapter 4, Learn by Doing: Building Two n8n Apps, shows you how to combine concepts from the previous chapters and use them to build multiple projects. Some of these projects will reinforce the concepts you learned earlier and others will introduce some new ideas. This will help you understand the kind of products that you can build using n8n. Finally, you will learn how to share and discover new workflows as well as participate in n8n's active community.

Chapter 5, Building Your First API Endpoints, will see you create blueprints for an API endpoint so that there's minimal friction when building the API; you'll also configure the Webhook node so that it can handle requests sent to the API and reply to them. You'll build an API endpoint in n8n based on the blueprints that you created earlier, and you'll secure your API endpoints by using the different authentication methods available in the Webhook node. Later, you'll also test your API to make sure that all the functionalities that you implemented work as expected.

Chapter 6, Powering Your API with a No Code Database, shows you how to work with no-code databases for data storage. You will learn about no-code databases, selecting a database for your project, and reading and writing to Airtable. You will also learn about some of the best practices when working with these databases. The concepts covered in this chapter will help you to use a data store for your projects to store user-generated data and build a complete product.

Chapter 7, Transforming Your Data inside a Workflow, explores how to manipulate data within workflows so that the APIs that you create can return data in a useful format. You will also learn about sharing data between workflows, working with arrays and JSON objects, merging datasets, and performing analytics and calculations.

Chapter 8, Utilizing the Bubble API in n8n, looks at how to communicate between Bubble and n8n, access Bubble's data using the Data API, use Bubble's workflows and interact with them using the Workflow API, and receive events and data from Bubble in n8n.

Chapter 9, Building the User Interface of the Application, dives into how to design responsive applications using the Bubble graphical user interface and how the look and feel of an application can change the user experience. You'll learn about underlying data structures and how to guide users to enter appropriate data into data structures. You'll discover how to identify errors in applications and workflows and proactively handle how errors are presented to users. Finally, you'll design a logging system to capture events and errors and analyze data captured in logs for application improvement.

Chapter 10, We've Only Just Begun, is where you'll see how far you've come in such a short period of time! In this last chapter, you'll have a look at what you learned from the book and get some help in finding and starting that next n8n project!

To get the most out of this book

Depending on the version that you are presently using, you may notice that what is on your screen may look a bit different than what is in the book. As we were nearing the end of the book, n8n updated their user interface, which explains why the examples here are different. With that being said, the scripts all continue to work exactly the same as described in the book and should work just fine for you.

Software/hardware covered in the book	**System Requirements**
n8n	Cloud or a local installation
Bubble	
Insomnia	

If you are using the digital version of this book, we advise you to type the code yourself or access the code from the book's GitHub repository (a link is available in the next section). Doing so will help you avoid any potential errors related to the copying and pasting of code.

Download the example code files

You can download the example code files for this book from GitHub at `https://github.com/PacktPublishing/Rapid-Product-Development-with-n8n`. If there's an update to the code, it will be updated in the GitHub repository.

We also have other code bundles from our rich catalog of books and videos available at `https://github.com/PacktPublishing/`. Check them out!

Download the color images

We also provide a PDF file that has color images of the screenshots and diagrams used in this book. You can download it here:

`https://static.packt-cdn.com/downloads/9781801817363_ColorImages.pdf`

Conventions used

There are a number of text conventions used throughout this book.

`Code in text`: Indicates code words in text, database table names, folder names, filenames, file extensions, pathnames, dummy URLs, user input, and Twitter handles. Here is an example: "Now, go to your bot and enter `/pokemon ditto`."

A block of code is set as follows:

```
<html>
<body>
<h1>From n8n with love 🤚</h1>
<b>Host:</b> {{$json["headers"]["host"]}}
</br>
<b>User Agent:</b> {{$json["headers"]["user-agent"]}}
</body>
</html>
```

Any command-line input or output is written as follows:

```
npm install pm2@latest -g
```

Bold: Indicates a new term, an important word, or words that you see onscreen. For instance, words in menus or dialog boxes appear in **bold**. Here is an example: "Next up, we have the **Executions** tab, which opens up a modal (popup) where you can view the executions of your different workflows."

> **Tips or Important Notes**
> Appear like this.

Get in touch

Feedback from our readers is always welcome.

General feedback: If you have questions about any aspect of this book, email us at `customercare@packtpub.com` and mention the book title in the subject of your message.

Errata: Although we have taken every care to ensure the accuracy of our content, mistakes do happen. If you have found a mistake in this book, we would be grateful if you would report this to us. Please visit `www.packtpub.com/support/errata` and fill in the form.

Piracy: If you come across any illegal copies of our works in any form on the internet, we would be grateful if you would provide us with the location address or website name. Please contact us at `copyright@packt.com` with a link to the material.

If you are interested in becoming an author: If there is a topic that you have expertise in and you are interested in either writing or contributing to a book, please visit `authors.packtpub.com`.

Share Your Thoughts

Once you've read *Rapid Product Development with n8n*, we'd love to hear your thoughts! Please `click here to go straight to the Amazon review page` for this book and share your feedback.

Your review is important to us and the tech community and will help us make sure we're delivering excellent quality content.

Section 1 – An Introduction to Your Toolkit

In this section, you will learn about the low-code space, install n8n, learn how n8n works, and build workflows for your specific use cases.

In this section, there are the following chapters:

- *Chapter 1, Introduction to No Code, n8n, and Bubble*
- *Chapter 2, Using the Editor UI to Create Workflows*
- *Chapter 3, Diving into Core Nodes and Data in n8n*
- *Chapter 4, Learn by Doing: Building two n8n Apps*

1
Introduction to No Code, n8n, and Bubble

A movement is slowly building in the tech industry. This movement sees ordinary computer users developing digital solutions and integrations with tools that abstract away computer code complexity. These tools, collectively referred to as no code tools, empower people who, previously, would never have been able to build solutions.

By the end of this chapter, you will have an understanding of no code and why it is becoming important to businesses and technology. You will also become familiar with n8n, a no code automation tool, and Bubble, a no code development platform.

This chapter will cover the following main topics:

- What is no code?
- Why does no code matter?
- An introduction to n8n
- How do people use n8n to solve problems?
- An introduction to Bubble

Technical requirements

For this chapter, you will need the following requirements:

- A computer running a Debian-based Linux distribution such as Ubuntu or Raspberry Pi OS
- Internet access to install n8n

What is no code?

Software has completely revolutionized how we work and live over the last half-century. In as little as half a century, expensive and bulky systems reserved for science and academia have evolved to become as ubiquitous as the kitchen sink, and often more affordable. We now collaborate with people on the other side of the planet as effortlessly as we talk with our neighbors over the back fence.

These systems have transformed modern society with their ability to automate complex tasks and manipulate data consistently. Instead of handwriting a letter, putting a stamp on it, and mailing it across the country to be received several days later, we compose an email. Thanks to the underlying infrastructure systems and automation that's been deployed over the last few decades, this information arrives in seconds.

But what happens when you want to do something unique or specific to your environment? What if you're going to do something niche? What do you do when there isn't an app for that?

Historically, you would learn a programming language and write a script or application to perform that task. Unfortunately, for many ordinary users, this is more than they are willing to take on. The idea of learning an entirely new language to save a few minutes each day does not feel like a favorable return on investment.

The other option is to hire a software developer to build the application. This option sounds like a logical one. But with the average annual salary of a software engineer in the United States being $110,000 (`https://www.indeed.com/career/software-engineer/salaries`), this may be too costly an option with little return on investment.

So, where does that leave us? Do you abandon the project, deciding that the savings are not worth the cost or the effort? Previously, that probably would have been the outcome of such a dilemma, but a third option has recently appeared on the horizon. It is at this point where **no code** steps into the picture.

Ordinary people use no code tools to create applications and services without learning a new programming language. They are often web-based tools that are intuitive to learn with minimal coding (if any) required. When coding is needed, the solutions are usually easily found on the product's community forums or built for a nominal fee by an entrepreneurial hobbyist.

This doesn't mean that having a coding background isn't helpful. Knowing the basic concepts surrounding computer programming such as how to write out the steps for a process from start to finish will help speed up the successes that a no coder experiences. But they are not strictly necessary.

Now that we have a better understanding of no code tools and the entire no code movement, it is essential to look at the ramifications of no code on the tech industry, business, and the average citizen. Is it something that completely changes how we, as a society, look at computers and technology in general, or is it just the latest tech fad that, in the end, will be quickly replaced with the "next big thing?"

Why does no code matter?

If you speak to certain developers today about no code tools, many of them will tell you that they are nothing but toys that don't have any real power. They may say that this is nothing but a fad and that if you genuinely want to develop applications, you need to use traditional programming languages.

Simultaneously, many businesses see these no code tools as a distraction, which takes their employees away from their tasks and duties. Rather than writing reports and filling out spreadsheets, these people waste their time making shiny phone apps that don't solve real problems or create bonafide value.

To be fair, these perspectives were once very valid. But in recent years, the no code landscape has significantly changed, and the gap between no code and traditional development tools has been shrinking. As the no code tools have been improving, the value that they bring to business has also been increasing.

Ironically, these two arguments are incredibly similar to statements that were used in the earlier days of computing. But they weren't talking about no code tools at that time. They often referred to flashy programming interfaces and upstart programming languages such as Visual Basic or JavaScript because real developers wrote in low-level machine language!

Many of these "distracting" toys are now valuable tools. For example, JavaScript was once just a tool used to create dynamic web pages. Now, tools written with JavaScript, such as Node.js, which is used by millions of developers worldwide, or jQuery, which can be found in over 75% of the websites on the internet, have proven how a once simple, niche idea can have a profound impact.

But there is a third perspective. These are the people who have embraced no code tools and see the potential that they truly offer. No longer is the business handcuffed to expensive developers or left to be dissatisfied with an off-the-shelf software package that performs some of the functions you require, but not really.

Instead, the worker who does the task every day has the power to use these no code tools to reduce their workload, increase their productivity, and reduce human error through data validation, automation, and eliminating repetitive tasks.

And because the worker is developing a solution, this end product can often quickly and accurately create value compared to a more expensive custom solution created by a developer who does not truly understand the role that this employee plays within the organization. No code tools are quicker to develop, cheaper to own, and often more efficient than development big brothers.

There is one final point about developing applications with no code tools. Because the barrier to entry is so much lower than complete development suites, tasks that would have otherwise been too cost-prohibitive to automate now become something that the employee can realize programmatically, by themselves.

Outside of daily business, we are starting to see a whole new class of entrepreneurs developing before our eyes on the internet. These are people who identify problems in day-to-day life and identify a win-win situation for both themselves and others. They are then using these no code tools to create applications and full-blown systems to solve these problems.

These entrepreneurs have also discovered that because the cost to develop these applications is so minimal and the time to market from initial concept to production is relatively short, a whole new way of building businesses has emerged. Rather than come up with an idea and put all of their time and effort into making that idea successful without knowing whether it will be successful in the end, these entrepreneurs are taking a different approach.

They will build several businesses simultaneously, some as many as 200 in one calendar year. This way, the entire market is their testing ground. They might try different strategies with different products, and instead of putting their heart and soul into one thing, they treat these businesses like cattle. The ones that grow strong and have a good return on investment are groomed and fed. The companies that don't produce or end up taking more time and energy than they are worth are taken out back and dealt with accordingly. No tears are shed for the failed business because the lost time/effort/money is minimal. The entrepreneur is too busy either growing the successful companies or returning to their no code tools to start the next one.

Now that we have a better understanding of the importance of no code tools in our modern businesses and day-to-day lives, let's take a closer look at a couple of these tools to understand some of their capabilities better.

An introduction to n8n

n8n is one of these no code tools. n8n's primary purpose is to connect different types of systems to share information in a meaningful way.

On its surface, that statement may sound like a relatively trivial thing that would have minimal impact on day-to-day work. But, when you start to dig a little deeper into how we do tasks and complete our work, you will quickly realize that there are many little jobs that we repeatedly do over and over again. If we can find a way for a system such as n8n to perform these little tasks for us, the cascade effect is more significant than just the time saved with that one task.

You also save the transition time between jobs. Many researchers have looked at context switching or how long it takes for us to transition from one task to another. They have discovered that it takes approximately 20 minutes to shift from doing one type of work to another. If you have one fewer job to do, that's one shift between tasks not needed, saving you that 20-minute transition time. There is also the time that's lost while waiting for something to happen.

If, for example, you have to fill out an online form with a set of information, such as a username and a password, and then wait for an email to come in after you have submitted the form, you have lost that time between hitting "submit" and waiting for that email to arrive. Instead, if you were to configure n8n to use an API connection to submit this data, not only would you not have to wait for the email to come in, you wouldn't even have to enter the data in a web form. n8n could perform this entire set of tasks on your behalf in a fraction of the amount of time it took you to fill out the form and wait for the email, all while you could have been doing something else far more critical. Now, imagine you have to do this 80 times a day. Wouldn't you rather spend that time figuring out how to get your computer to do it for you? I think you would.

n8n is a web-based tool. n8n can run on a computer system as small and inexpensive as a Raspberry Pi. It can connect to several hundred different systems with **nodes** designed explicitly for those systems and thousands more that use standard REST API interfaces. It can also manipulate the data that it receives and sends out so that the information in each system does not have to be identical when moving data around. It can also produce some analytics, providing insights into the information that's returned from the processed data. And because it can interface with almost any REST API-based system, this means that the only limits to its capabilities are the limits of those other systems. Do you want any of them to be able to create 3D images? There's a system with an API for that. Do you want n8n to be able to remix music on the fly based on the color patterns it sees coming in from a webcam? There's a workflow for that. Would you like n8n to automatically balance your business's books to provide you with a real-time dashboard of the fiscal health of your organization? There are ways to do that as well. If there is a way to connect n8n to a system, then n8n can inherit those powers.

While each of these nodes is very powerful in and of itself, the true potential of n8n is evident when you begin to connect them. Each node is designed to perform a specific set of actions. Upon completion, it then passes this information on to the next node. How does a node know where to send its data to next? It simply looks at the connections between the nodes.

Each node has an **input** (left) side and an **output** (right) side. If a node's output connects to a second node's input, the second node receives the outcome of the first node as its input. A collection of two or more nodes is known as a workflow.

These workflows can then be saved in the n8n system and activated to run on their own in the background when you do not have the n8n Editor UI open:

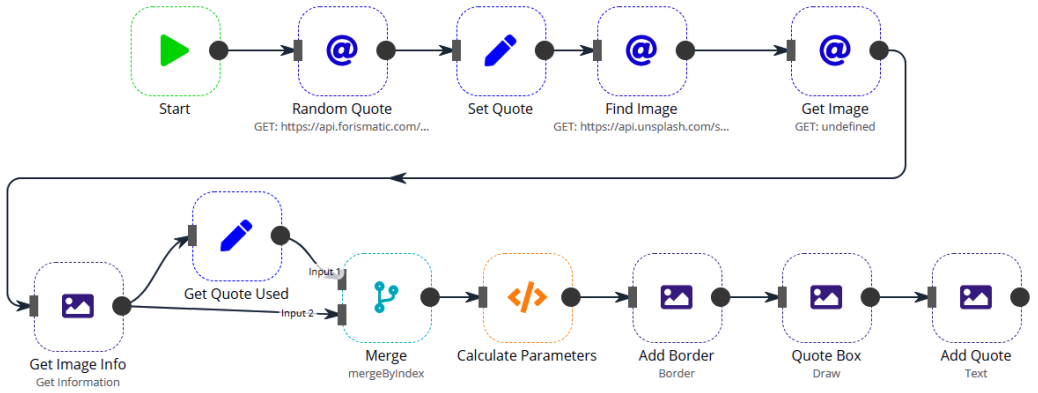

Figure 1.1 – Sample n8n workflow

But how do these workflows know how to run when I don't tell them to run? When the workflow is on my screen in the n8n Editor UI, I can click the **Execute Node** button to make the workflow run. But if I'm not available to click the button, how does it know to run? Now is the time for the **trigger nodes** to shine!

In general, the trigger node starts a workflow. The first one is the **Cron** node. The Cron node gets its name from a Linux service that executes commands at specifically scheduled times. The Cron node does the same thing. You specify a time – for example, every day at noon – and the workflow connected to this Cron node will run at that specific time.

What happens if an event occurs between the times you have the Cron node configured to run? Several other trigger nodes come into play. Most of these nodes listen for events coming in from a specific system. When it detects these events, it will take the information that's been sent to it and process it through the workflow connected to that node.

New trigger nodes are added to n8n regularly. But there is still a chance that the specific system you want to listen to does not exist as a node. What do you do now? The answer is to use a webhook node. A webhook node is a generic version of these customized trigger notes. It allows any system to submit an HTTP request to an address on the network to send information to n8n when it's using this node. The webhook node opens up n8n to all kinds of other possibilities.

Now, wouldn't it be great if there was a node much like the webhook node but instead of listening to any system that can make an HTTP request, it could talk to any system listening for HTTP requests? As you may have guessed, that is what the HTTP request node does. This node gives any system the ability to talk with almost any other system with a **REST API**.

The REST API is an internet standard used for data communication between two systems. More and more developers are writing REST APIs for their new systems and services. Older systems and services are often retrofitted with new APIs to make them more accessible to other data systems.

Now that we have been introduced to n8n and its benefits, let's install it!

Installing n8n

n8n is operating system agnostic, meaning it will run on Mac, Windows, and Linux operating systems. n8n can do this because it runs on top of the Node.js JavaScript library, enabling developers to run JavaScript applications as services.

> **Note**
>
> Just prior to this book going to press, n8n released a new version called n8n Desktop. It installs much more like a typical application rather than a service.

Installing n8n varies from one operating system to the next, but people generally have the most difficulty with Linux. This is primarily because Linux is the clear choice for installations such as cloud service providers or inexpensive home lab scenarios such as Raspberry Pi installations. We have chosen to provide detailed installation instructions for Linux systems, specifically Debian-based systems, because of these challenges. Some examples of Debian-based systems include Ubuntu and Raspberry Pi OS. You can find instructions for other operating systems on the n8n website at `https://docs.n8n.io/#getting-started`.

Prerequisites

To successfully install n8n on a Linux computer, you will require the following items:

- A Debian-based Linux operating system
- A regular user account with `sudo` access
- A modern web browser
- An internet connection

1. Updating the operating system

It is crucial to have an updated operating system to make sure that you are installing the latest version of the software. Perform this by following these steps:

1. Log into your computer with a standard user account.

2. At a Terminal, enter the following command to update the system:

```
sudo apt install update -y &&sudo apt install upgrade -y
```

3. When you're prompted for a password, enter your user password.
4. The system will then install several packages and updates.
5. Once the installation is complete, proceed to the next section.

2. Installing the prerequisites

n8n requires several different packages to run correctly. Follow these steps to install these packages:

1. At a Terminal, enter the following command to install the prerequisites:

```
sudo apt install build-essential python -y
```

2. If you're prompted for a password, enter your user password.
3. The system will then install several packages.
4. Once the installation is complete, proceed to the next section.

3. Installing Node.js

As we mentioned earlier, Node.js is the core upon which n8n is written. Without Node.js, n8n will not run. These instructions will install version 14.x:

1. At a Terminal, enter the following command to configure the installation source for Node.js 14.x:

```
curl -sL https://deb.nodesource.com/setup_14.x | sudo -E
bash -
```

2. If you're prompted for a password, enter your user password.
3. Enter the following command to install Node.js:

```
sudo apt install nodejs -y
```

4. If you're prompted for a password, enter your user password.
5. Once the installation is complete, proceed to the next section.

4. Configuring the Node.js environment

For n8n to work properly, it needs to be installed globally on the system. By default, Node.js does not allow you to install n8n globally without configuration changes. Follow these steps to configure the Node.js environment properly:

1. At a Terminal, enter the following commands to create a new hidden folder in your home directory:

```
cd ~
mkdir ~/.nodejs_global
```

2. The following commands configure the npm package manager to use the new folder for global installations:

```
npm config set prefix ~/.nodejs_global
echo 'export PATH=~/.nodejs_global/bin:$PATH' | tee
--append ~/.profile
source ~/.profile
```

3. Once the installation is complete, proceed to the next section.

5. Installing n8n

The system is now ready to install n8n:

1. At a Terminal, enter the following command to start the n8n installation:

```
npm install n8n -g
```

2. The installation will now proceed to install several packages, along with the n8n application.

> **Note**
> While n8n is installing, you may encounter several warnings. It is normal to receive them, and you should not be alarmed.

3. Once the installation is complete, proceed to the next section.

6. Using PM2 to run n8n as a service

n8n typically runs in a Terminal, but as soon as you exit the Terminal or reboot your computer, n8n will no longer run. To solve this issue, we can use an application called PM2 to run n8n as a service. Let's install it:

1. At a Terminal, enter the following commands to install PM2:

```
cd ~
npm install pm2@latest -g
```

2. Once PM2 is installed, use PM2 to start n8n:

```
pm2 start n8n
```

3. Next, configure PM2 to start n8n automatically if your computer is restarted:

```
pm2 startup
```

4. The system will now prompt you to execute a command line. Make sure that you enter it exactly as it is presented on the screen.
5. Finally, save the new PM2 configuration:

```
pm2 save
```

7. Opening the n8n Editor UI

n8n should now be running. You can check that it is running properly by following these steps:

1. If you don't know your computer's IP address, enter the following command in a Terminal:

```
hostname -i | awk '{print $1}'
```

2. It will return a value like `192.168.0.1`.
3. Open a web browser and go to `http://<IP address>:5678`. So, for our example, you would go to `http://192.168.0.1:5678`.

4. The n8n Editor UI will open with an empty workflow, as shown in the following screenshot:

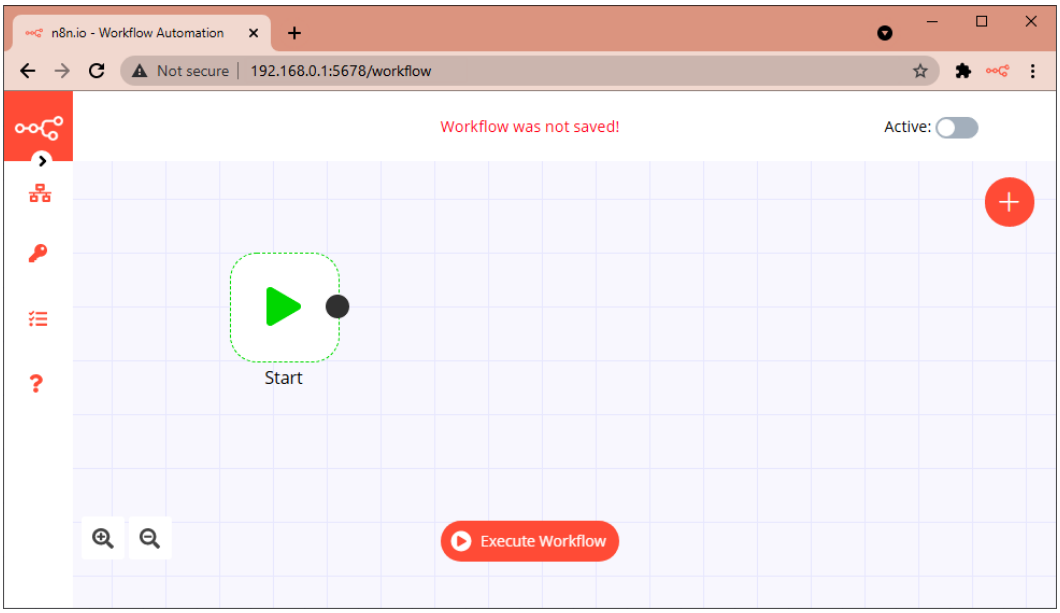

Figure 1.2 – n8n Editor UI

With n8n successfully installed, it's time to start building some valuable solutions. But, sometimes, the most challenging part of building a solution is deciding what to make in the first place. Let's explore some ways in which people have used n8n to resolve real-world problems.

How do people use n8n to solve problems?

While n8n has many different capabilities, its primary role is connecting systems and preparing the data correctly for each system to ingest.

An excellent way to understand the capabilities of n8n is with a few real-world examples. The following are setups that are being run by real companies, giving them a technical advantage over the competition.

CRM call recording access

One of my former employers had purchased a new Linux-based phone system and a separate Raspberry Pi-based system to record all of the phone calls that came into the organization, along with Windows-based software to manage the recordings. They also had a Windows-based **customer relationship management** (**CRM**) system for keeping track of sales and inventory, with a Microsoft SQL Server backend.

The sales division head wanted the sales team to retrieve all the calls that had been made or received from a customer based on date and then click on a button or link to listen to the recording.

The final solution grabbed all of the call log data from the phone system and temporarily stored it in a text file. n8n would then monitor this file for changes and import the data. At this point, it would pull in the call recording information, correlate it with the call logs, and write this data to Microsoft SQL Server. The IT team then modified the CRM to look for this information in the new table and display it in the CRM.

Because n8n is both data- and system-agnostic, it effectively bridged these radically different devices and enabled efficient, accurate communication among all of them. Without n8n, this would have required a series of different scripts, scheduled jobs, file exports, network shares, and constant monitoring. n8n allowed the IT team to perform these same tasks in one single system using only the tools that came with it.

Goomer pivots during COVID-19

Goomer started as a restaurant ordering software company in Brazil. But when COVID-19 struck in early 2020, the government shut down all restaurants, and Goomer's clients were now struggling to stay in business. Goomer's business was suddenly in trouble.

Goomer decided to become a delivery application, shifting all of their development teams to designing and engineering the new program. Their technical teams did not have time to focus on automation or deployment tools. It was at this point that they decided to turn to n8n to build out this automation.

They chose n8n because it was significantly cheaper and more flexible than other hosted solutions. Because they could run it locally without any surprise billing, they could focus on deploying their tools and running their business without having to worry about scaling or prohibitive pricing.

Goomer uses n8n to manage their business workflow by connecting systems such as Airtable, Coda, and HubSpot. n8n automatically reacts to events in these systems to update data and information in the other systems without relying on any user intervention.

n8n sails the seven seas

n8n isn't limited to dry land. Maranics takes human manual processes and digitizes them to increase the quality of the process and reduce workloads. After using other automation solutions for several years, Maranics started to migrate their automation workflows to n8n for many clients, including several cruise ships.

Maranics used n8n to easily connect different databases such as MongoDB and Postgres to other systems with standard REST APIs, which would have been significantly more difficult using other solutions.

Since n8n can be installed locally in a business (or a ship), there is no need for a persistent internet connection when all the data is stored on systems locally. Plus, there is no need for the data to leave the network or local business since all of the data can be processed right there on-premises.

These are just a few examples of how people have taken n8n and built workflows to make their processes more efficient, less prone to human error, and available 24/7.

It's fascinating to see how people have used n8n to connect systems. But, occasionally, there is no all-in-one system out there that does what you want. Other no code tools step in at this point to fill in the gap. They provide various services such as web frontends or data storage. Then, n8n is used to connect it to the rest of the world. Bubble is one of those tools, and we will be using it extensively throughout this book.

An introduction to Bubble

Because n8n's forte is integrating different systems, the development team at n8n has not focused on user interface development. The development team knows that entire companies focus on excellent no code tools for building that frontend and are happy to leave that to them. (Although, we will show you how to make a web frontend entirely using n8n later in this book.)

Since n8n's focus is not on frontend design, we chose to go with Bubble as the no code development tool for this purpose. Specifically, we decided to use Bubble because of the following reasons:

- It has a simple and intuitive development interface.
- It is a popular and well-known no code tool.
- It has APIs for both data access and workflow activation.
- It has limited workflow functionality.
- It has little integration with other systems.

These factors make it an excellent tool to use. But it is not the only tool that we could have used to interact with n8n. There are very few no code tools that n8n cannot connect with, using a custom-designed node or a REST API.

Summary

This introductory chapter explored no code and the people who make it a driving force for change in present-day businesses. We also discussed some of the significant benefits of using tools such as n8n and provided some real-world examples of how companies have leveraged no code's flexibility.

We also took some necessary first steps toward understanding n8n a bit better. You learned how to install n8n on a Debian-based Linux system so that you can get started right away with using n8n at your home or business at no cost to you.

Finally, we introduced you to Bubble, a no code tool that we will be using to develop frontends for many of our example applications. We'll be able to use Bubble and n8n in tandem to create full-fledged solutions that are user-friendly (Bubble) and interact with various systems and services.

Now that you have a better understanding of no code, n8n, and Bubble, it's time to dig deeper into how n8n works and build some of our first workflows. We will do this in the next chapter.

2

Using the Editor UI to Create Workflows

In this chapter, you will learn how to use n8n's **Editor UI**. This will help you find the different functionalities of n8n. You will also learn about the different kinds of nodes in n8n and how to use them. You will then learn about building workflows and dealing with dynamic data to handle different kinds of scenarios. You will finally create your first workflow in n8n.

This chapter will cover the following main topics:

- Introduction to the Editor UI
- Exploring the regular and trigger nodes
- Expressions – using dynamic data
- Workflows – putting it all together
- Creating your first workflow – Hello World

Technical requirements

This is a list of technical requirements to prepare before continuing with the chapter:

- Have n8n installed.
- Ensure n8n is running and the Editor UI is open.
- Create an account on Telegram.

Introduction to the Editor UI

The Editor UI is a graphical interface that allows you to create automations using a node-based approach. n8n takes its inspiration for node-based visualization from the film and television industry, where many tools have a node-based system. Let's take a look at what we have here:

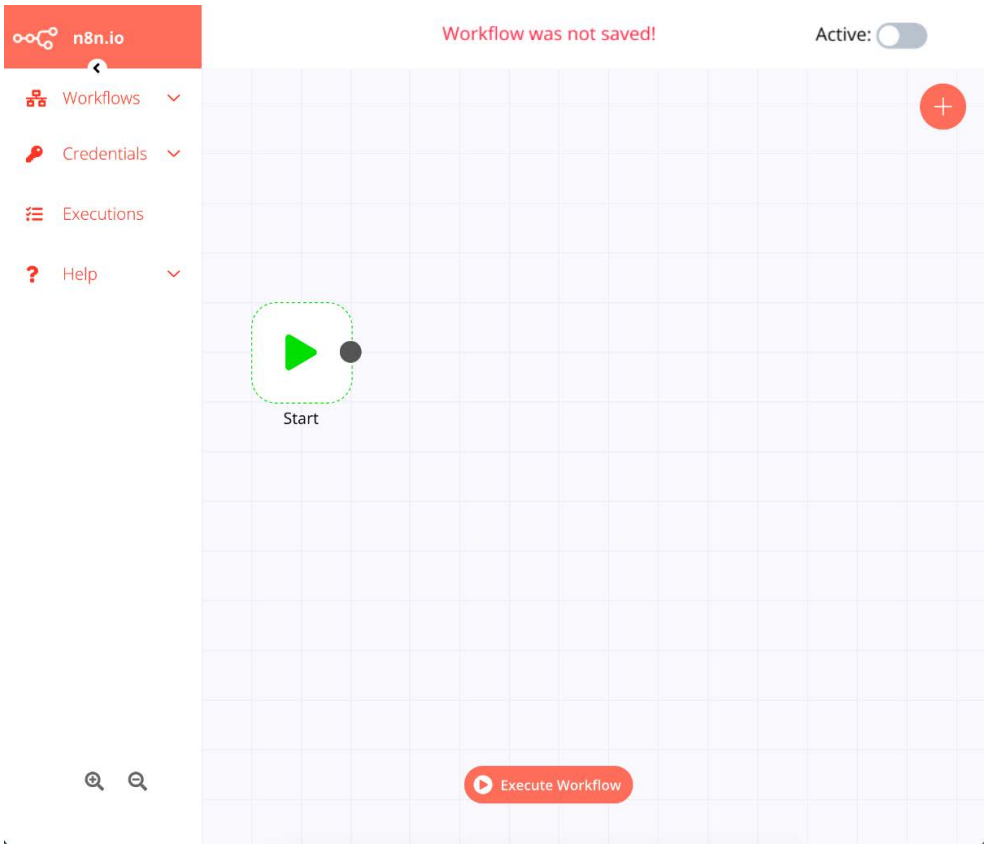

Figure 2.1 – The Editor UI in n8n

Let's start from the top left. You can expand the menu by clicking on the > button beneath the n8n logo.

For now, we'll get ourselves familiar with the interface so that we can quickly find our way through n8n. We'll dive deeper into specific sections, such as **Executions**, as we progress through the chapters in this book. Let's take a look at the menu bar on the left.

First of all, we have the **Workflows** menu. In the menu, we have the following options:

- **New**: Create a new workflow.
- **Open**: Open an existing workflow.
- **Save**: Save changes to the current workflow.
- **Save As**: Save the current workflow.
- **Rename**: Rename the current workflow.
- **Delete**: Delete the current workflow.
- **Download**: Download the workflow as a JSON file.
- **Import from URL**: Import a workflow from a URL.
- **Import from File**: Import a workflow from a JSON file.
- **Settings**: Configure the settings for the current workflow.

Please note that some of these options will be grayed out since we haven't saved the workflow.

Next, we have the **Credentials** menu. In this menu, we have two options:

- **New**: Create a new credential.
- **Open**: Open an existing credential.

n8n allows you to connect to many different applications, services, and APIs. A lot of these require credentials to authenticate yourself. n8n enables you to encrypt and save these credentials in its database so that they can be quickly reused when building workflows.

Next up, we have the **Executions** tab, which opens up a modal (popup) where you can view the executions of your different workflows. You can also filter the executions by their name and status.

Finally, we have the **Help** tab, which lists resources that will be useful for you:

- **Documentation**: A link to the n8n docs. The n8n docs contain detailed information on each node, example workflows, and references.
- **Forum**: n8n has a very active and friendly community. If you get stuck on anything, feel free to drop a question there and someone will help you resolve it.
- **Workflows**: This page lists workflows that have been submitted by the community. It's a great place to gain some inspiration for your next automation.
- **About n8n**: This option opens up a modal with details about n8n's version, a link to the GitHub repository, and the license.

At the bottom left, you'll see options to zoom in and out of the canvas. The canvas is the grid of boxes in the Editor UI where you'll be adding the nodes to create workflows. You'll also notice that all new n8n workflows have a **Start** node.

At the bottom center of the Editor UI, you'll notice a button labeled **Execute Workflow**. This is useful for manually executing the workflows that you have created in n8n. Workflows can run in two ways:

- **Manual executions**: These are useful for testing your workflows while you are building them or for one-off executions. An example of one-off execution can be a workflow that migrates all the data from a CSV file to Google Sheets. You can manually execute a workflow by clicking on the **Execute Workflow** button.
- **Automated executions**: Once your automations are ready, most of the time, you'd want them to run at regular intervals or when a specific event occurs. For this to happen, you'll need to activate a workflow. You can do so by clicking on the **Active** button at the top left, which will change the state of the toggle from inactive to active. Please note that you'll need to first save the workflow before being able to activate it.

Finally, we have the + button on the right side of the Editor UI. Clicking on it will reveal the **Nodes** panel. You can use this to add new nodes to the canvas and build out your workflow. You'll notice that it has three categories: **All**, **Regular**, and **Trigger**. Let's learn a bit more about nodes in n8n to understand what these sections mean.

Exploring the regular and trigger nodes

Nodes are the building blocks of workflows in n8n. Nodes can connect to applications, services, and APIs, and do anything that is possible with Node.js in general. Each node performs tasks based on its design. Upon completion, the data is passed on to the next node. This data is the result of the work performed by the configured task in the previous node.

Each node has an input (left) side and an output (right) side. If a node's output connects to a second node's input, the second node receives the outcome of the first node as its input.

> **Note**
> We will cover specific nodes in greater detail later on in the book.

From an abstract point of view, there are two types of nodes in n8n:

- **Regular nodes**: Regular nodes are useful for things such as handling **Create, Read, Update, and Delete** (**CRUD**) operations with applications and APIs, transforming data, and pulling information from the internet. Some examples of regular nodes include the following:
  - **Airtable node**: It can read, add, update, and delete data from an Airtable table.
  - **Function node**: It can execute JavaScript code, most often to manipulate workflow data.
  - **Bubble node**: This node allows you to create, delete, update, and get objects from Bubble.

  You can find the whole list of regular nodes in the latest version of n8n by heading over to the **Apps & nodes** page of the n8n website (`https://n8n.io/integrations`) and clicking on the **Regular** tab (see *Figure 2.2*):

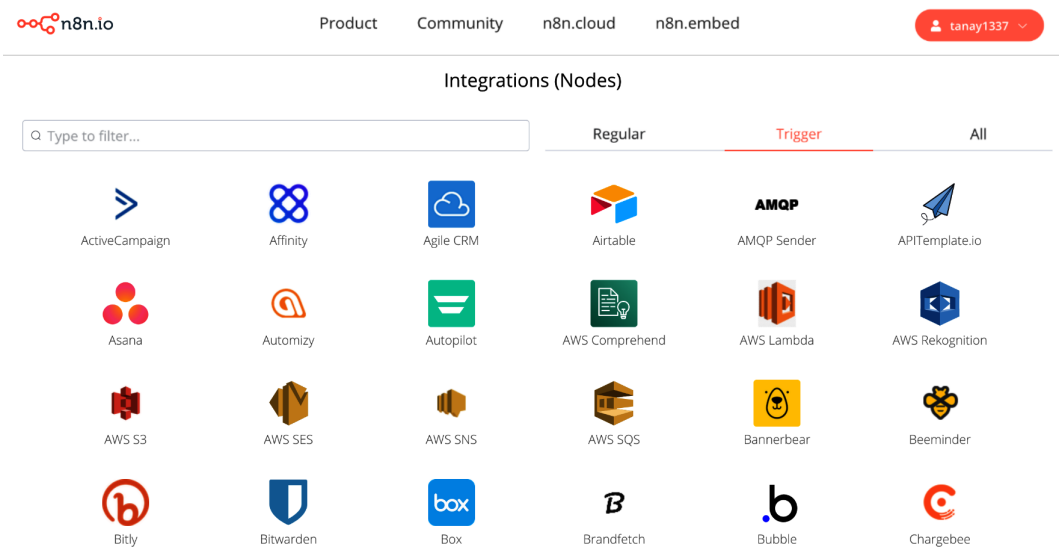

Figure 2.2 – Some of the regular nodes that n8n comes with

- **Trigger nodes**: Trigger nodes start the execution of a workflow. These nodes can start a workflow based on events such as time-based intervals or events from external systems. You need a trigger node in a workflow if you want it to execute automatically. You cannot activate workflows unless they have a trigger node. A couple of examples of trigger nodes include the following:
  - **Cron node**: This node can be configured to activate a workflow every minute or every hour, or specify using custom Cron expressions.
  - **Telegram trigger node**: This node can be configured to activate a workflow every time a Telegram bot receives a message.

  You can also find an exhaustive list of trigger nodes on n8n's **Apps & nodes** page by clicking on the **Trigger** tab (see *Figure 2.3*):

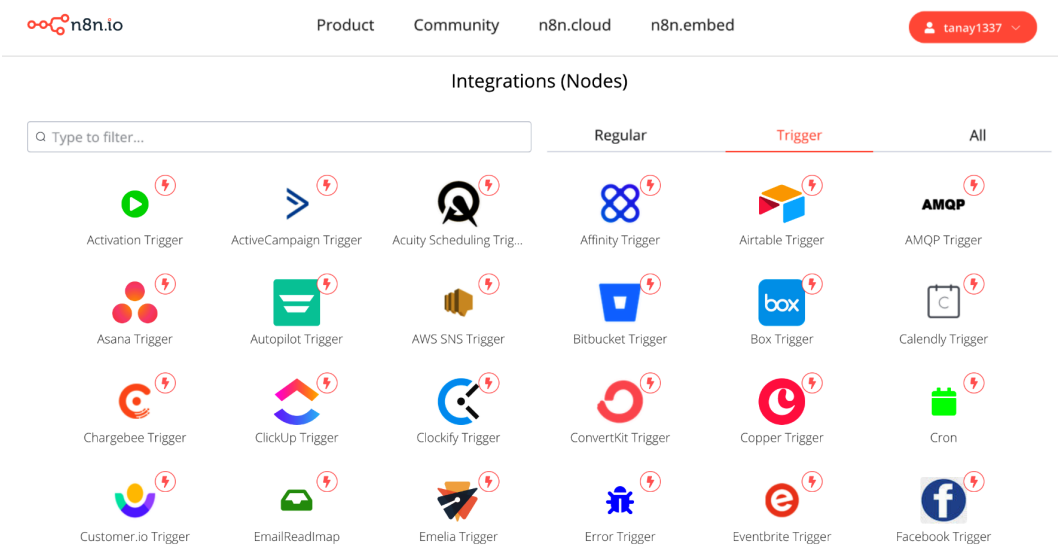

Figure 2.3 – Some of the trigger nodes that n8n comes with

Some of the nodes in n8n are referred to as core nodes. As of the time of writing, there isn't a distinction in the Editor UI for these nodes. Core nodes could be both regular or trigger nodes. These are nodes that are more general purpose, such as the following:

- **Webhook node**: Can be used to receive webhook responses
- **Function node**: Can be used to manipulate workflow data

- **Cron node**: Can be used to trigger workflows at specific intervals
- **HTTP Request node**: Can be used to make HTTP requests to a web page or an API endpoint

We'll dive deeper into the core nodes in n8n in *Chapter 3*, *Diving into Core Nodes and Data in n8n.* To put this concept to practice, follow these steps:

1. Let's go back to the Editor UI. First of all, click on the **Start** node so that it's highlighted. Doing this ensures that the new node that you add to the canvas will automatically connect to the highlighted node.

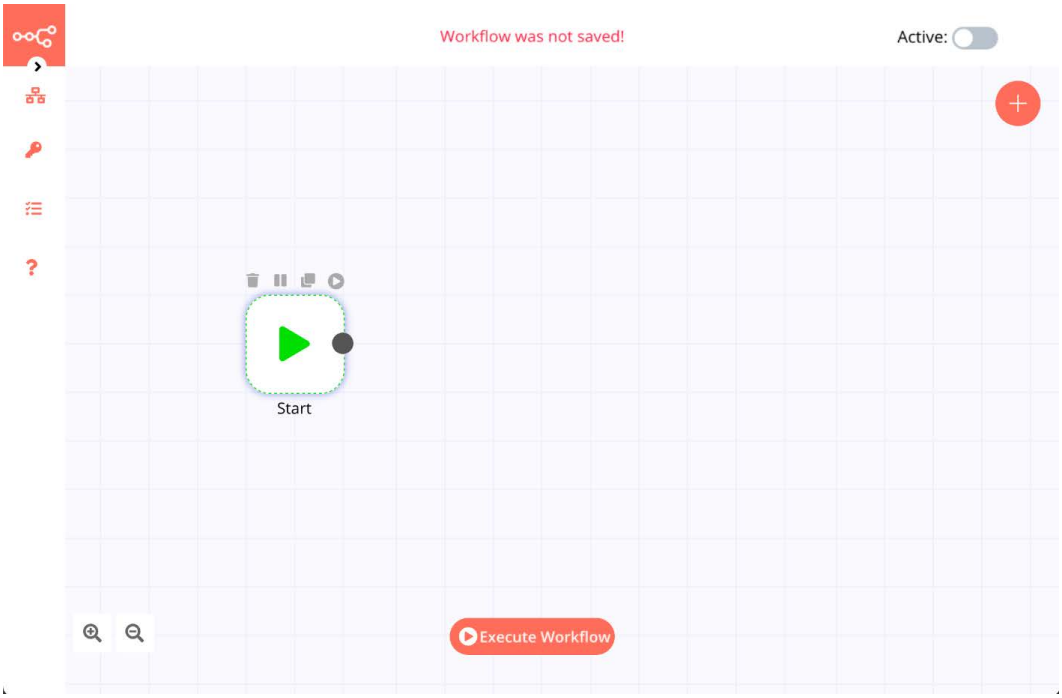

Figure 2.4 – Highlighting the Start node by clicking on it

If you didn't do that, no worries. You can always connect two disconnected nodes by clicking on the circle in front of the node and dragging it to the rectangle of the next node.

2. Let's add a node to the canvas by clicking on the + button in the Editor UI.
3. Type `Hacker News` into the **Nodes** panel and select the **Hacker News** node.

4. Clicking on the node will add it to the canvas and open the node to be configured. Select **All** for the **Resource** field and click on the **Execute Node** button at the top right of the node panel.

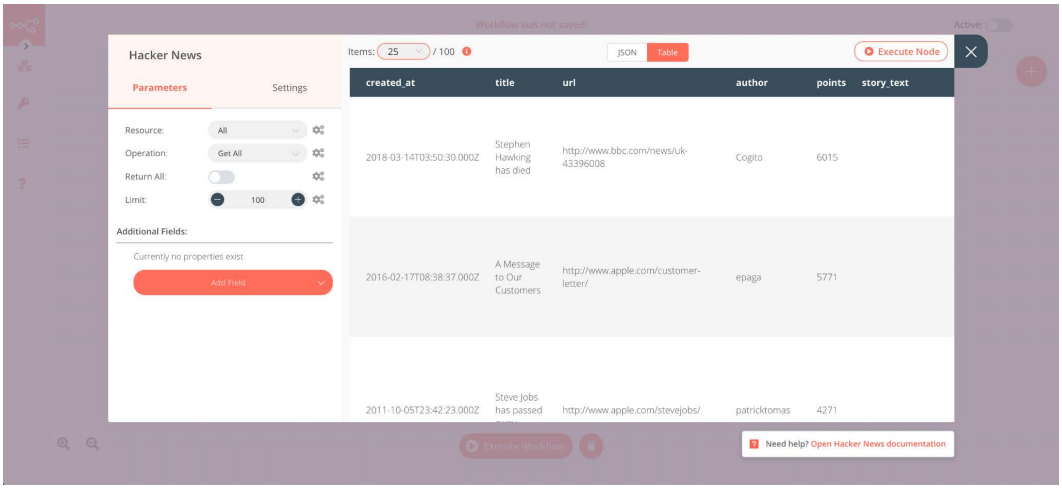

Figure 2.5 – Output of the Hacker News node

You will notice that the node returns 100 news articles from Hacker News. It produces a variety of information for each news item. Let's take a few minutes to understand what we have in the node panel.

Clicking on any node in n8n will open up the node details view, which allows you to configure the node, execute it, and look at the data that was either received or generated by it.

Let's start with the top left of the node panel. It says **Hacker News**, which is the name of the node. If you add another **Hacker News** node to the workflow, it will be called **Hacker News1**, and so on. You can rename a node so that it's easy to gain context later on. To do that, click on the node name, type in the new name, and click on the tick icon. Let's rename it to `Get news` for now.

Under the name of the node, we have two tabs:

- **Parameters**: This tab contains parameters to configure the task. Most of the nodes have the **Resource** and **Operation** fields. They are a way to bundle together the wide variety of functionalities offered by the various applications and APIs.

  To understand this better, let's take the example of a CRM where you can store the data of individuals and companies. In n8n, a node for this CRM would have **People** and **Company** as resources. For each of the resources, it might then have create, read, update, and delete as the operations.

Underneath these two fields, you would have the fields that are required, for example, **Name**. Like the last name, any optional fields are always bundled together under the **Additional Fields** section, which keeps the UI uncluttered.

- **Settings**: This tab contains a couple of options, such as adding notes, and a couple of other settings that we'll cover later on in the book as they become relevant.

In the top middle of the node panel, you will see two tabs:

- **JSON**: Clicking on this tab will showcase the data received from **Hacker News** in JSON format.

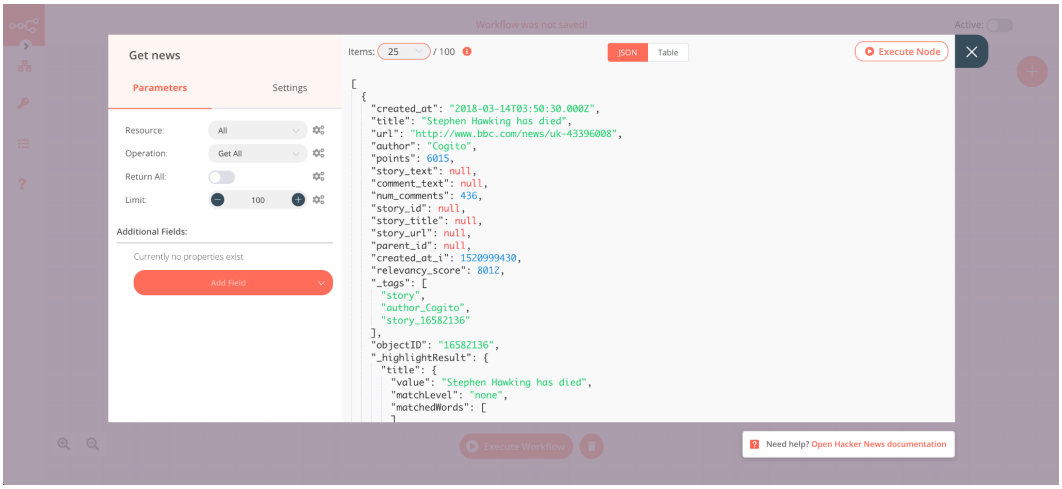

Figure 2.6 – Viewing the output by clicking on the JSON tab

- **Table**: The table view is the default view for displaying data in n8n.

At the top right of the panel, you'll see the **Execute Node** button. This button enables you to execute this specific node manually without re-triggering the entire workflow or executing the following nodes in the workflow. This is very useful for testing the workflows while building them step by step. We then have an **X** button next to it. You can click on it to go back to the canvas in the Editor UI.

Finally, at the bottom of the node panel, you'll see a link to the node's documentation that you have opened up. Click on it to reach the node documentation, which contains example workflows with that node along with some FAQs.

When working with nodes, you will often need to reference data between nodes. This data would usually be dynamic. For example, weather data might change every hour. Let's learn about how we can reference dynamic data in n8n using expressions.

Expressions – using dynamic data

Consider this scenario: you only want the name of the news article and the URL as part of your workflow data because the rest of the data returned by the Hacker News node is irrelevant to you. We can filter out the workflow data in a couple of ways, but we'll use the **Set** node here. The **Set** node is one of the code nodes in n8n that helps you to configure workflow data To create a Set node for the aforementioned scenario, follow these steps:

1. Add the **Set** node to the canvas and connect it to the **Get news** node.

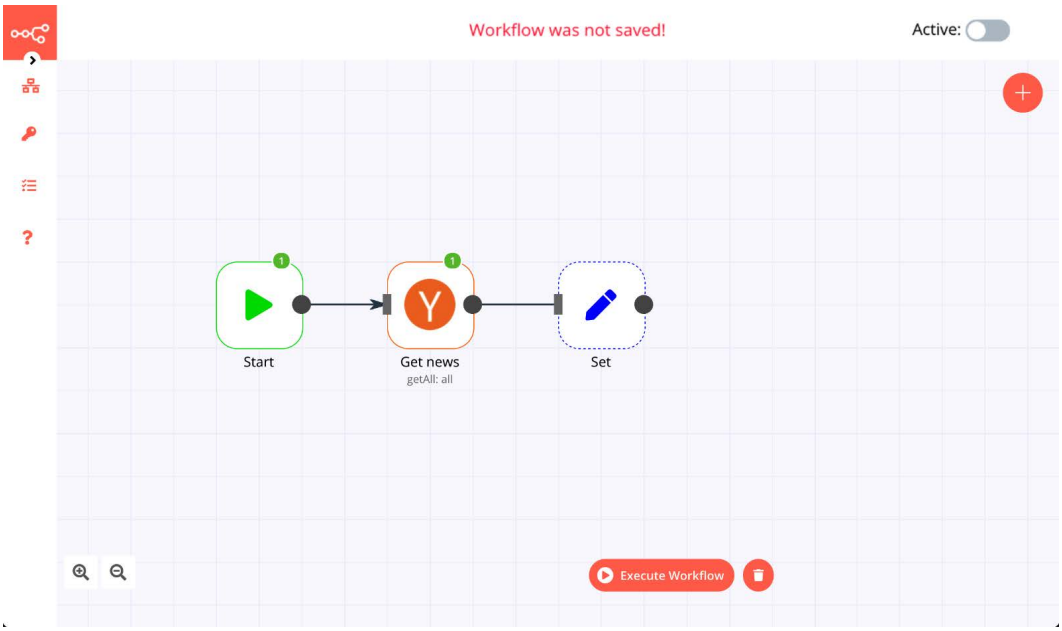

Figure 2.7 – Adding the Set node to the Hacker News node

2. Now, open the **Set** node by double-clicking on it. Toggle the **Keep Only Set** field to true (green). It removes all incoming workflow data and only appends the new values configured in the **Set** node.
3. Click on the **Add Value** button and select **String**. You'll notice that we now have two fields: **Name** and **Value**. These two fields design the JSON data structure.
4. Delete `propertyName` from the **Name** field and enter `Title`.

Now, the value of the title will be different each time the node iterates over the 100 items that it will receive from the previous node. It is also possible that those 100 values will be completely different as new articles appear on Hacker News and so on.

Because of that, the **Value** field needs to be dynamic since the value keeps changing. You can achieve this in n8n using expressions.

5. To add an expression to a field, click on the gears icon next to the field and click on **Add Expression**. Perform this step with the **Value** field. These actions will open up the Expression Editor.
6. On the left-hand side, you'll notice the **Variable Selector** section. Click on the current node and go through the nesting to find the title. It should look like this:

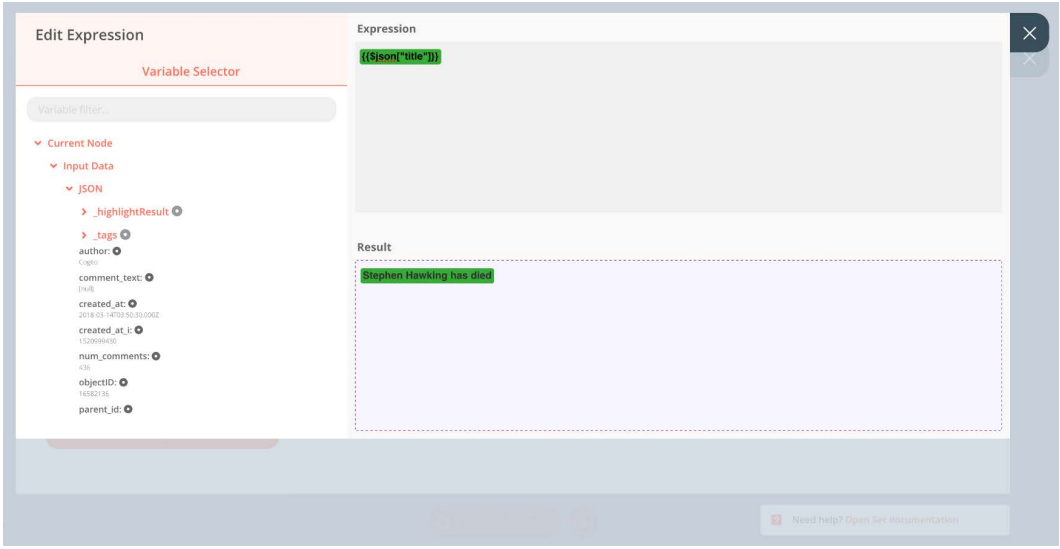

Figure 2.8 – Selecting the title of a post using the Expression Editor in n8n

If the data does not show up for you, check the following:

- This node and the previous node are connected.
- The previous node has been executed.

We'll perform the same steps for the URL and then click on the **Execute Node** button for the **Set** node.

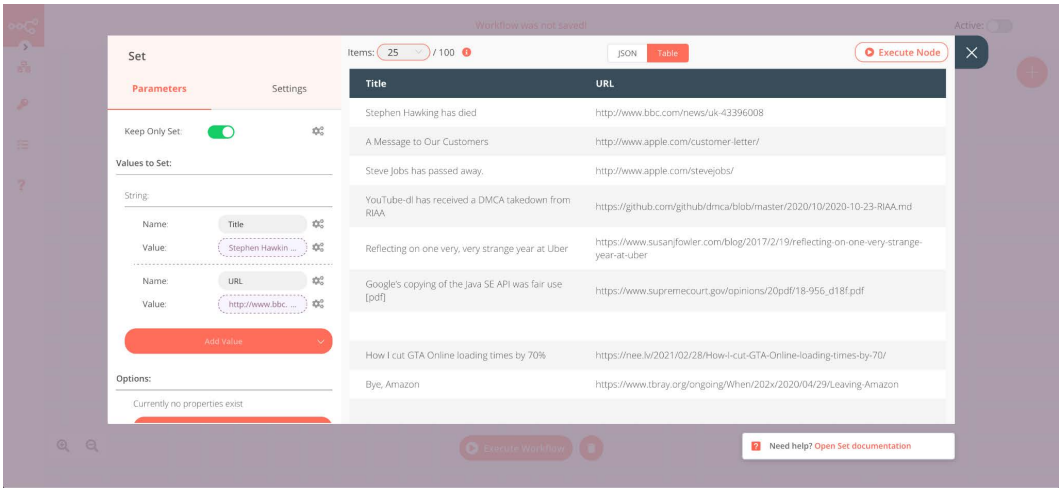

Figure 2.9 – Output of the Set node

We now have only the relevant data that we wanted in the workflow, thanks to the **Set** node and expressions.

Expressions are a powerful feature in n8n. You can use them to reference data from the workflow, other nodes, the environment, and even self-generated data. Let's say we want to add a random ID to each of the news articles. Add a numerical value in the **Set** node, enter `ID` as the name, and click on **Add Expression** for the **Value** field.

Delete the `0` and enter the following: `{{ Math.floor(Math.random() *1000) }}`.

Expressions can execute JavaScript between double curly braces, and here we've used it to generate three-digit random numbers. Let's say that we want the IDs to be prefixed by `ID_`. Edit the expression so that it looks like this: `ID_{{ Math.floor(Math.random() *1000) }}`. Click on the **Execute Node** button to see the result.

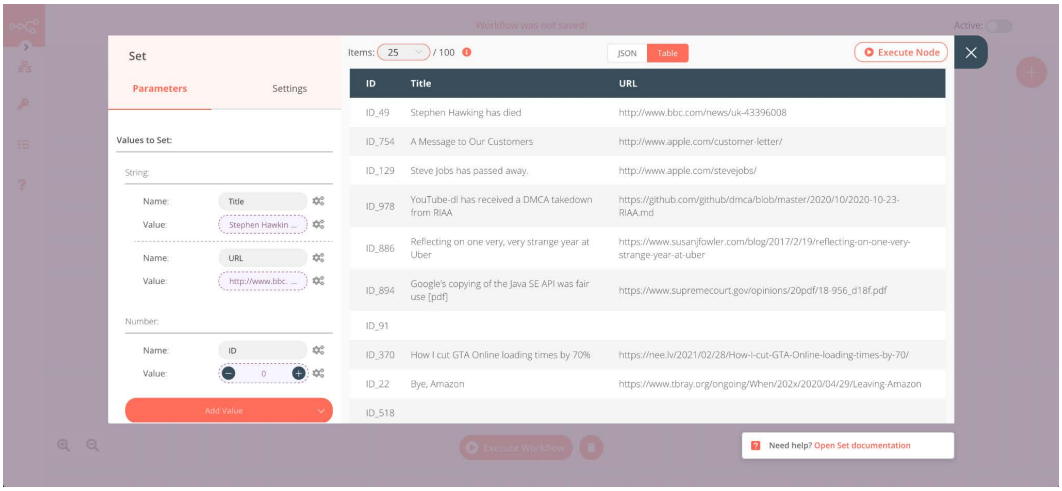

Figure 2.10 – Output of the Set node after adding the ID

There are many other things that you can do with expressions, and we'll introduce more concepts throughout the rest of the book.

Now that we know about expressions, let's learn more about workflows in n8n to understand what they are.

Workflows – putting it all together

Workflows are a collection of nodes in n8n. Workflows can range anywhere from two nodes to hundreds of nodes, with workflows calling sub-workflows. These workflows can then be saved in n8n's database and activated to run on their own in the background even when you do not have the n8n Editor UI open.

While each of these nodes in n8n is very powerful in and of itself, the true potential of n8n is evident when you begin to connect these nodes. In n8n, you can join one node to many other nodes. Workflows don't necessarily have to follow a linear structure.

Real-world problems or tasks are usually made up of a series of steps. n8n is powerful because you can model your complex processes and tasks as a workflow, a series of nodes that each performs a step toward completing the bigger process.

Let's now save the workflow created by clicking on the **Workflows** icon at the top right and clicking on **Save As**. Let's call it `My first workflow` and press *Enter*.

Workflows are JSON objects. You can take a look at the underlying JSON in one of two ways:

- Click on the **Workflows** icon and click on the **Download** button to download the JSON file.
- Press *Ctrl* + *A* on the canvas to select all the nodes in your workflow, press *Ctrl* + *C* to copy them, head over to a text editor, and paste the JSON by pressing *Ctrl* + *V*.

Here's how the JSON for the workflow that we created appears:

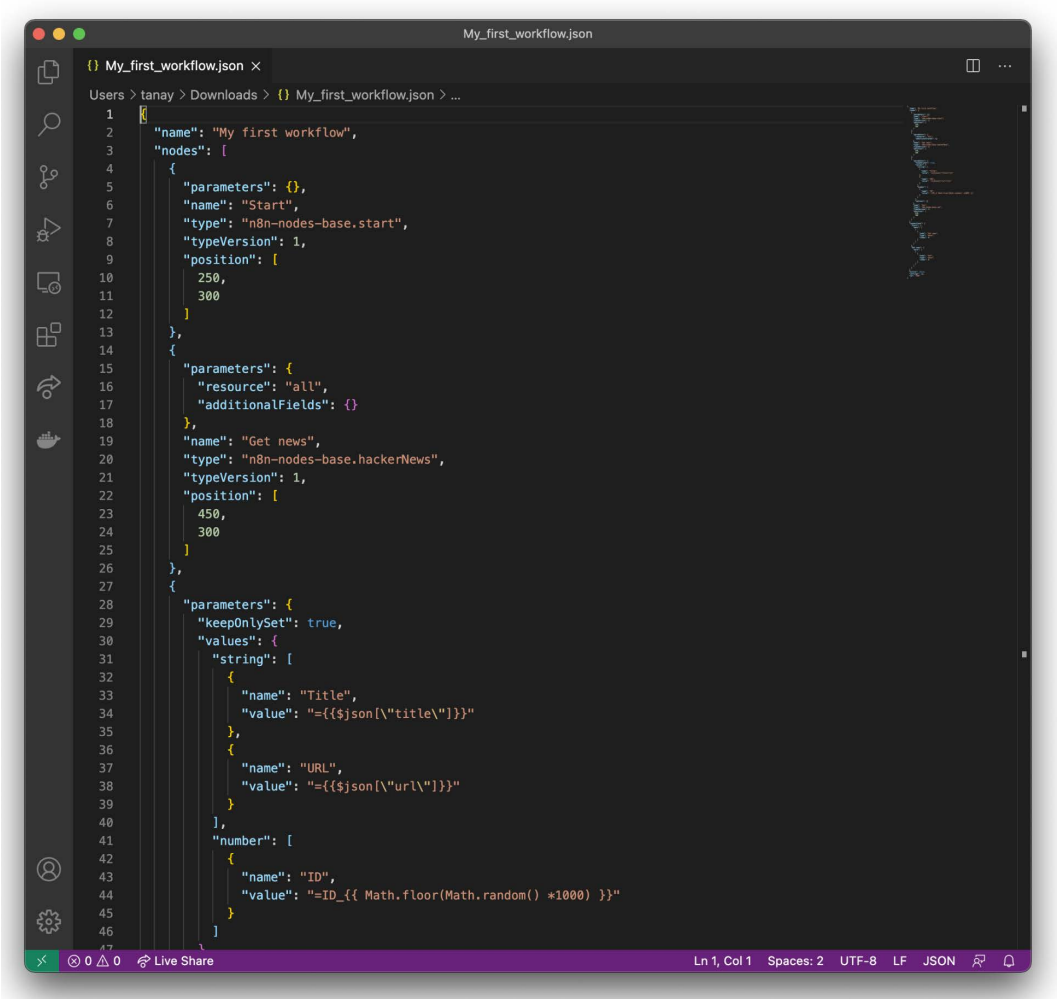

Figure 2.11 – JSON for the workflow that we created

If you look at the JSON file, you'll see that it contains information about the different nodes present in your workflow, how they are connected, and the set parameters. In the preceding screenshot, you can notice the title, URL, and ID that we configured in the **Set** node and the expressions that we added.

You can then share these files with your friends and colleagues so that they can load them on their n8n instances and run the workflows that you created. They can either import the workflow as a file or the JSON and paste it into their Editor UI.

Now that we have learned about workflows, let's create our first workflow in n8n, which will send us a cocktail recipe in Telegram every day at 6 P.M.

Creating your first workflow – Hello World

Now that we are familiar with n8n's Editor UI, nodes, and workflows, let's create a new workflow that allows us to send a random cocktail recipe to Telegram every day. Follow these steps:

1. Create a new workflow on n8n by clicking on the **Workflows** icon and then clicking on **New**. Since this is an automation that runs every day, we'll need to add a trigger node.
2. Click on the **+** button, click on the **Trigger** tab, and select the **Cron** node.
3. From the **Cron** node's details view, click on **Add Cron Time** and change **Hour** to **18**. By doing this, we ensure that the workflow runs every day at 1800 hours.
4. Save the workflow and name it **Hello World**. Now, the workflow will be triggered at 1800 hours for the default time zone in n8n. n8n's default time zone is **New York**.
5. Let's change the time zone for this specific workflow by clicking on the **Workflows** icon and selecting **Settings**. There, you can choose your time zone.

6. Once you have done that, click on the **Save** button and return to the Editor UI.

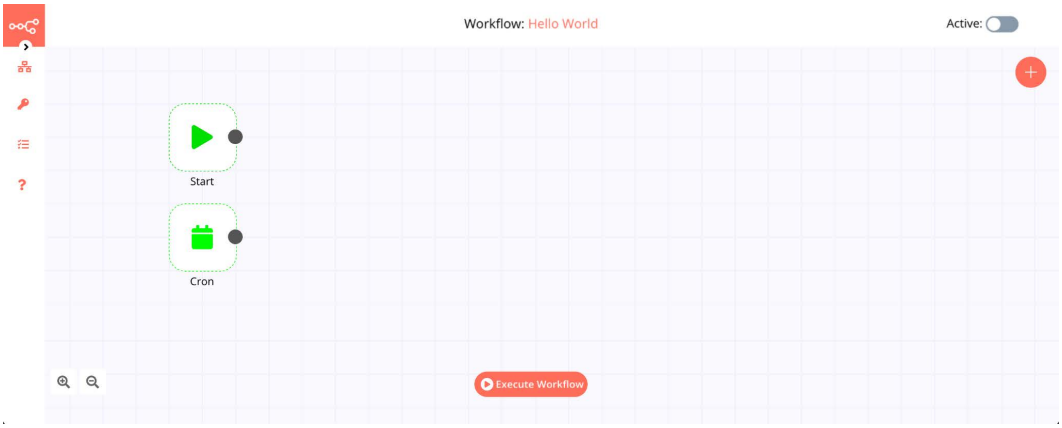

Figure 2.12 – Adding a Cron node to the Editor UI and saving the workflow

7. Now, highlight the **Cron** node by clicking on it and then add the **HTTP Request** node to the canvas. Make sure that it is connected to the **Cron** node.

   We'll use the **HTTP Request** node since the cocktail API does not have a node in n8n. The **HTTP Request** node allows us to make HTTP requests and enables us to make API calls to services that don't have a node in n8n yet. We'll be making a call to the random endpoint of the CocktailDB API.

8. Enter `https://www.thecocktaildb.com/api/json/v1/1/random.php` in the **URL** field of the **HTTP Request** node and click on the **Execute Node** button. You will notice that it returns details about a random cocktail.

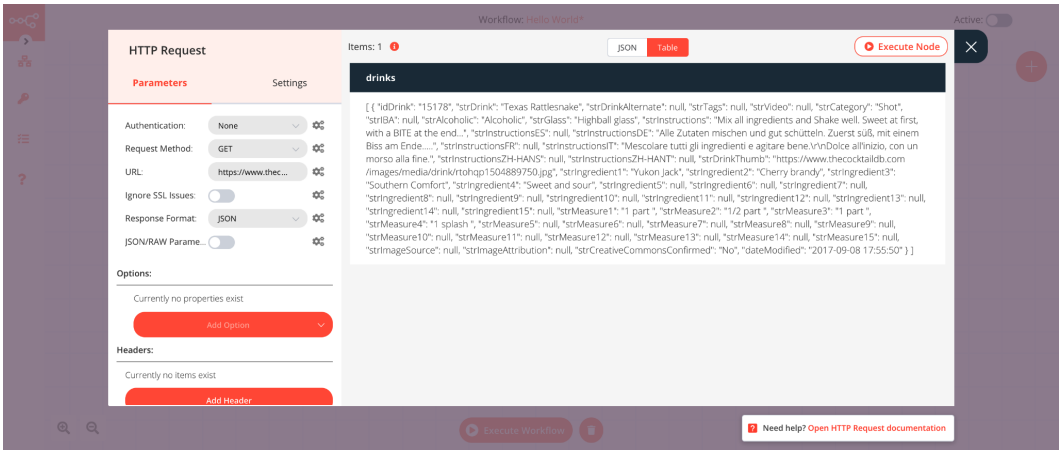

Figure 2.13 – Output of the HTTP Request node after making a request to the random endpoint of the CocktailDB API

Now, we'll have to send this data to Telegram. To do that, we'll first need to create a Telegram bot. You can use either the Telegram mobile app, web app, or desktop application for that. In the example screenshot (*Figure 2.14*), I am using the desktop app for macOS. To do that, follow these steps:

1. Search for `BotFather` with a blue and white "verified" symbol next to its name. To verify, click on the BotFather username.

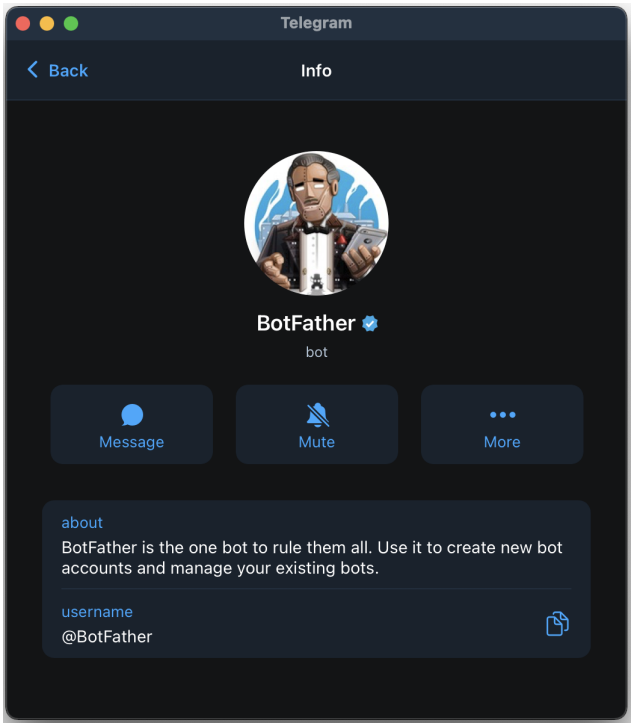

Figure 2.14 – BotFather in the Telegram application

2. Enter the `/newbot` command and follow the instructions to create your bot. You can name it as you please, but the username needs to be unique.
3. Once completed, it will give you an access token for the HTTP API. Copy that; we'll need that in n8n. Next, click on your bot link provided by BotFather and click on **Start**.
4. Now head over to n8n and add a **Telegram** node. Make sure that the input of the Telegram node is connected to the output of the **HTTP Request** node. You will notice that the node has a section called **Credentials**.
5. Click on the **Telegram API** field and select **Create New** from the dropdown.

6. The **Create New Credentials** modal will open up. Enter a name for the credentials. You can name it anything you want. I called it `Daily Drinks Bot` to differentiate it from my other Telegram bots at a glance.
7. Now, paste the access token you copied from BotFather in the **Access Token** field, and click on the **Save** button.
8. Now, we need the chat ID. To find that, open a new tab in your web browser and go to `https://api.telegram.org/bot<YourBOTToken>/getUpdates`.

   Don't forget to replace `<YourBOTToken>` with the access token that you got from BotFather. If you see nothing on that page, send a message to your bot and open the URL again. You can then copy the chat ID from there and paste it into the **Chat ID** field in n8n.

   Now, we have to craft the message for the **Text** field. We'll use expressions for this. Feel free to craft the message as you like. Here's how mine looked:

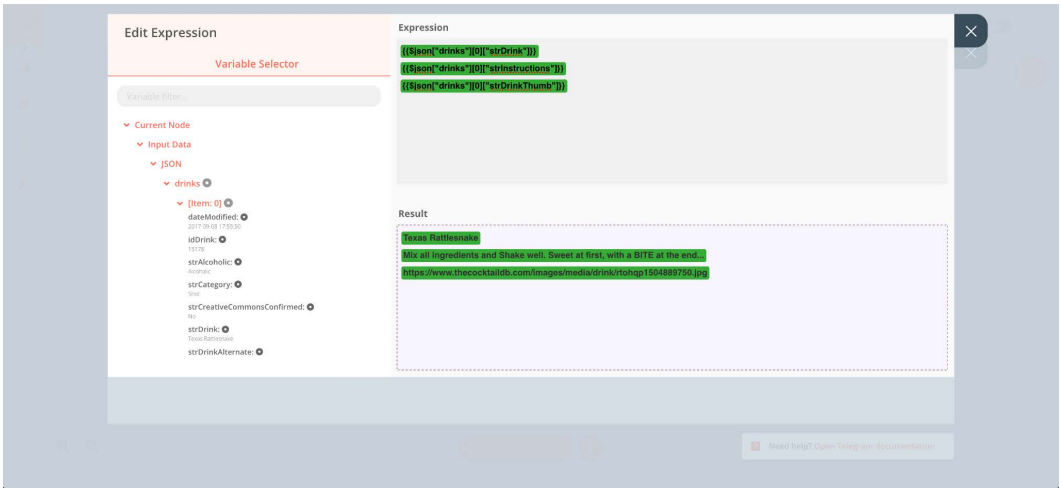

Figure 2.15 – Using the Expressions Editor to specify the content of the Telegram message

9. Head back to the Telegram node panel and click on the **Execute Node** button. You should now see a message from your bot on Telegram. Here's how mine looked:

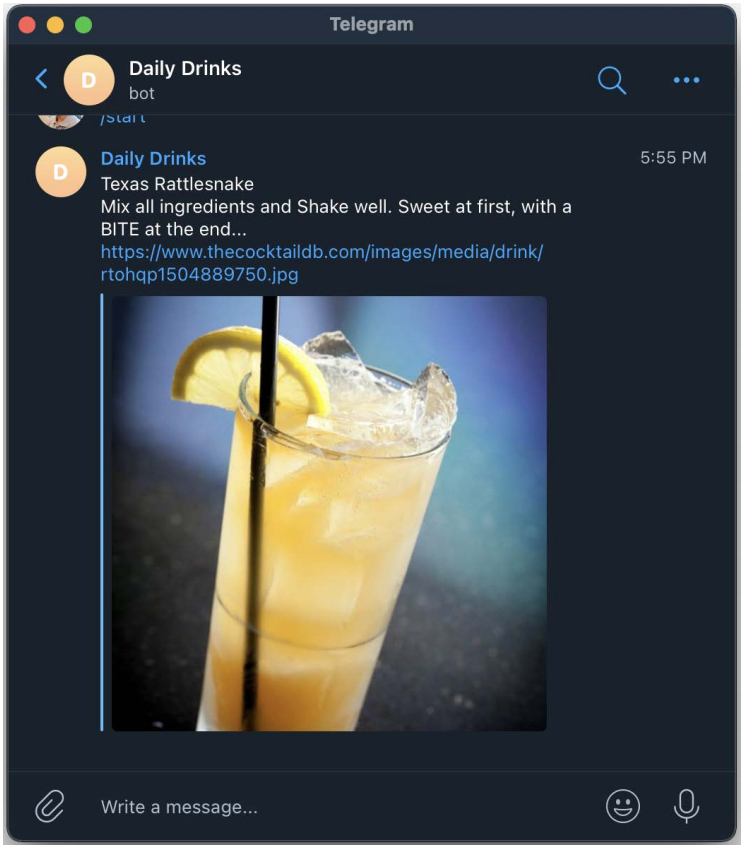

Figure 2.16 – Message sent by our n8n workflow via the Telegram bot

Now that our bot is up and running, we will have to activate the workflow to get a new cocktail recipe every day at 1800 hours.

10. Return to the canvas in the Editor UI, click on the **Activate** button at the top right, and select **Yes**, then activate and save! My final workflow looks like this:

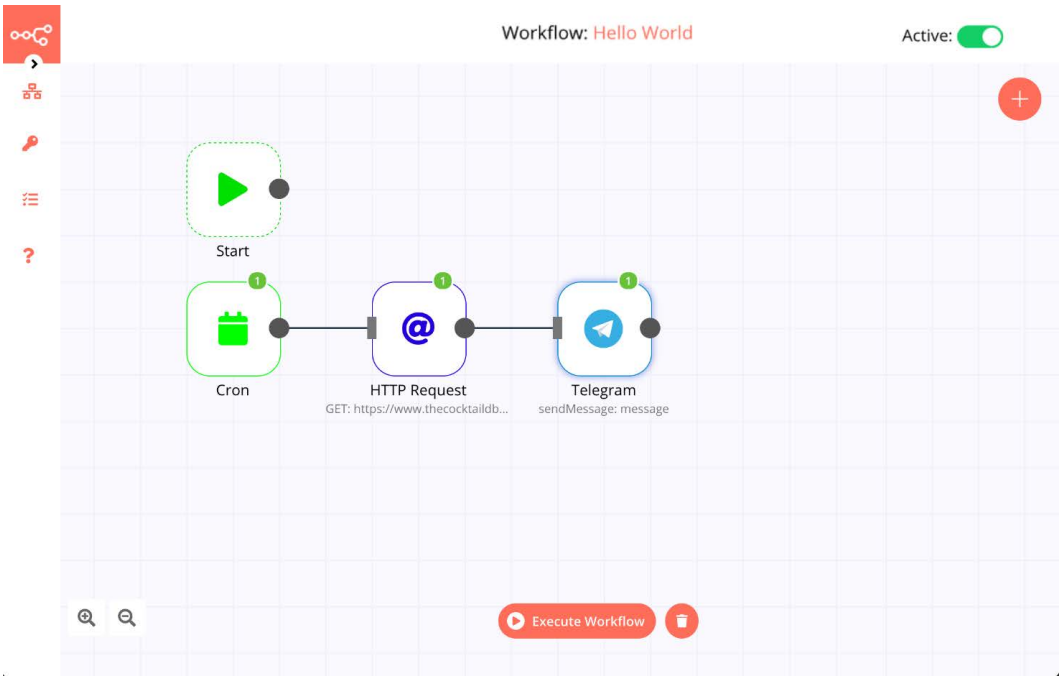

Figure 2.17 – Final workflow that has been activated

As long as n8n is running, you will now get cocktail recipes every evening. Congrats on creating your first n8n workflow!

Summary

In this chapter, we learned about n8n's Editor UI. We then learned about the two types of nodes in n8n. We then covered expressions to reference dynamic data in n8n workflows and learned more about workflows in n8n. Finally, we created our first workflow in n8n by putting together knowledge of the Editor UI, nodes, expressions, and workflows. We'll use the principles that we've learned throughout this chapter to build workflows that will handle the backend of our products. An understanding of these topics will help with building workflows of any complexity in n8n.

In the next chapter, we are going to dig deeper into the core nodes that make n8n so powerful. We will then take a look at how n8n workflows share data between themselves and learn how to access that information.

3
Diving into Core Nodes and Data in n8n

If you have ever been to a construction site just as they are starting to build a home, it can sometimes be challenging to envision what the final home will be like, and the future owners are probably very excited to get into the home and make it their own. If they had their way, they would probably skip this part of the home build and focus on all the details such as paint color, room layout, and furniture placement.

But if all home builders were to do this, none of their homes would last very long. They would not have a proper foundation upon which to sit, and the frame of the home would quickly fail.

The same thing can be said for learning to design computer programs. If a new developer does not learn the foundational parts and concepts of a programming system, then the application build goes very slowly and tends to be of poor quality. But unlike with a traditional computer programmer, it is not necessary to understand complex syntax or coding structure to become proficient at developing with a no-code solution.

It is crucial to properly understand the core nodes and how data is structured in n8n. Without these foundational concepts, you will not get very far with n8n.

This chapter covers the following topics:

- Introduction to the data structure in n8n
- Function node—Custom JavaScript in workflows
- The items array
- HTTP Request node—Talk to any **application programming interface** (**API**)
- Webhook node — Handling real-time events
- Working with arrays and **JavaScript Object Notation** (**JSON**) objects

By the end of the chapter, you will have learned how to do the following:

- Use n8n's data structure to manipulate and transform data inside workflows
- Use JavaScript inside your low-code workflows to unlock custom functionalities
- Call **REpresentational State Transfer** (**REST**) APIs from inside your workflows using the `HTTP Request` node
- Handle real-time events using the `Webhook` node and trigger workflows based on this data
- Work with arrays and JSON objects inside n8n, understand when to use what kind of data format, and transform data inside the workflows

Technical requirements

- You should have installed n8n
- n8n should be running, and the Editor **user interface** (**UI**) is open
- You have access to the GitHub repository, which can be found here: `https://github.com/PacktPublishing/Rapid-Product-Development-with-n8n`

Introduction to the data structure in n8n

I love traveling to different countries and experiencing life from a different perspective. Different cultures and customs fascinate me.

But unfortunately, I often find myself running into trouble because I am only fluent in a single language. Without having a language in common, it is easy to misunderstand what someone else is trying to communicate to me.

n8n also has its version of a language that it uses to communicate between nodes. This "language" is known as JSON and is a simple text format that is easy for both computers and humans to read. Using JSON, n8n can transfer both text and non-text (known as **binary**) information.

The developers at n8n designed each node in the workflow to receive and output data in this specific JSON format. This standard data structure allows n8n to chain these nodes together in infinite workflow combinations to produce a wide variety of different solutions.

It is essential to have a strong understanding of JSON in general and, specifically, how n8n uses JSON. This section talks about how JSON represents information and two different ways of grouping it—objects and arrays. We also delve into how n8n uses JSON to communicate inside a workflow and store the two main types of data inside a workflow—JSON and binary data.

In the following code snippet, you will find a sample of JSON-formatted information. This example describes a 2021 red car with automatic transmission:

```
[
  {
    "vehicle": {
      "type": "car",
       "colour": "red",
       "year": 2021,
       "automatic": true
    }
  }
]
```

JSON syntax

There are a few syntax items to cover, making it easier to understand these JSON files. These items can be broken down into the following categories:

- **Values**—Data or information
- **Key-value pairs**—Name of the information and the information itself
- **Objects**—Groups of key-value pairs
- **Arrays**—Groups of values

We will be learning more about these in the following sections.

Values

A value is a piece of data represented in JSON. A value can be:

- A string (a series of alphanumeric symbols)
- A number (standard numeric value)
- A JSON object (see the *Objects* section)
- An array (see the *Arrays* section)
- A Boolean (`true` or `false` value)

Key-value pairs

The first piece of JSON information to understand is key-value pairs. These are made of a field name in double quotes and a value separated by a colon.

For example, `"fruit": "apple"` is a key-value pair, with `"fruit"` being the key and `"apple"` being the value.

You can reference a key in n8n to retrieve the value that is paired with it.

Objects

An object is a group of key-pair values enclosed within curly brackets and separated by a comma. For example, `{"name": "Jill", "age": 27, "certified": true}` is an object with three keys (`name`, `age`, and `certified`), and each key has a value (`Jill`, `27`, and `true`).

Their key can reference values in the object.

To take this a step further, let's imagine we have the following JSON object:

```
{"user1":
     {"name": "Jill", "age": 27, "certified": true},
 "user2":
     {"name": "James", "age": 43, "certified": false}
}
```

In this example, `user1.name` would be `Jill` and `user2.name` would be `James`.

Arrays

An array is a group of values enclosed within square brackets and separated by a comma. For example, `[ "zero", "one", "two" ]` is an array with three values.

Arrays are similar to objects, except they do not have keys, so their index references the values.

An index is the position of a value in the array. The index value starts with zero (0) and increases by one for each portion in the array. In our example, these are the indexes of the array and their values:

- `0`—`"zero"`
- 1—"one"
- 2—"two"

n8n JSON structure

The JSON that is passed between nodes in n8n is a particular structure. It is made up of an array of at least one object.

That object has either one or two sub-objects inside it. The two object keys are `json` and `binary`.

The json object

The `json` object is a required object. Inside it is all of the JSON data that you will see as the result of a node execution.

The contents of the `json` object are flexible and can be a mixture of other objects, arrays, and key-value pairs.

For example, if you have a set note that sets the key `colour` to a `red` value, the node output will appear like this in JSON:

```
[ { "colour": "red" } ]
```

But when this information is stored in the n8n JSON format, it is passed between nodes, looking like this:

```
[
  {
    "json": {
      "colour": "red"
    }
  }
]
```

This way, the nodes understand that the information is meant to be for them and identify the information as test information. There is also an optional binary section to go along with the json section, and that will be covered a bit later.

The following diagram illustrates the format in which data is passed between nodes. It represents the framework for the data:

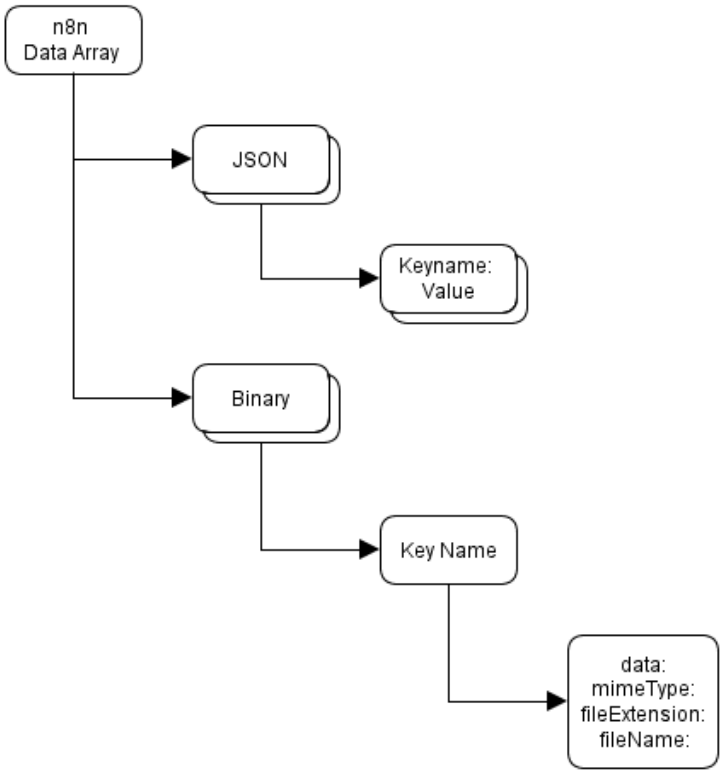

Figure 3.1 – n8n data structure

The entire set of data that is passed between nodes is built into a JSON array. Inside that array, there are two JSON objects—one called `JSON` and another called `Binary`. The `JSON` object contains key-value pairs representing text data. Meanwhile, the `Binary` object contains binary information (think of this as a file). Along with the actual data of the `Binary` object, there is some metadata such as `mimeType` (this is the type of file the data contains), `fileExtension`, and `fileName`.

The binary object

The second object in the n8n JSON data structure is the `binary` object. The `binary` object is an optional component since not every JSON dataset contains a binary element. But when it is provided, this object contains information that generally represents a file in a filesystem.

But if you are to include this object in your n8n JSON data, it has a particular structure. Inside the `binary` object is a key named whatever you wish (for our example, we will call it `binaryKeyName`). The value associated with this key is another object. This object is made up of up to four key-value pairs:

- `data` (required)—Base64-encoded binary data or unique data ID reference
- `mimeType` (optional)—The type of data stored in the `data` value based on standard mime types
- `fileExtension` (optional)—The extension that the file representing the information in the `data` value has
- `fileName` (optional)—The name of the file describing the data in the `data` value
- `path` (optional) - The location of the file on the system

When the `binary` object is set in the n8n JSON data, you will see an extra **Binary** tab at the top of the open node. It will contain the information that is in the `binary` object, as illustrated in the following screenshot:

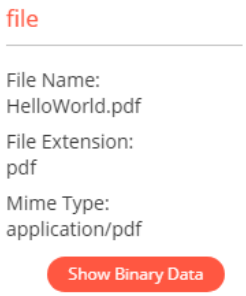

Figure 3.2 – Binary data in n8n Editor UI

In the next section, we are going to take a look at the `Function` node. It allows you to create custom code in JavaScript in case there is no node to perform the exact action you need. This is where understanding the n8n data structure becomes extremely important for manipulating information using JavaScript code.

Function node – Custom JavaScript in workflows

Sometimes, the perfect n8n node for the action you want to complete simply doesn't exist. This makes sense because there are an infinite number of different actions that could take place. This is why n8n created the `Function` node: so that we would have a way of creating our own custom actions, and we are going to learn how to do that next.

The `Function` node is the most versatile node in the n8n toolbox. It can execute JavaScript to manipulate data output from other nodes and then output it in the standard n8n data structure. You can see a screenshot representation of it here:

Figure 3.3 – Function node

But the flexibility of the `Function` node does require you to be able to use some JavaScript. Unlike many other nodes with several options and parameters to pick and choose from, the `Function` node only has a single parameter—the JavaScript code field. It is in this field that you will do all of your work.

The JavaScript code field contains, surprisingly, JavaScript. You will use this programming language to do all data manipulations within the `Function` node. While you don't need to be a JavaScript expert, there is some value in getting to know this language better. For what we will be talking about in this book, a basic understanding of JavaScript will suffice.

The first time you open up the JavaScript code field in a new `Function` node, you will notice that there are already several lines of code:

```
// Code here will run only once, no matter how many input items
there are.
// More info and help: https://docs.n8n.io/nodes/n8n-nodes-
base.function

// Loop over inputs and add a new field called 'myNewField' to
the JSON of each one
for (item of items) {
  item.json.myNewField = 1;
}

// You can write logs to the browser console
console.log('Done!');

return items;
```

This gives you an excellent example of how the `Function` node works. This script assigns the value of `1` to the `myNewField` key in every item in the JSON object of the n8n data structure. It then writes to the web browser console that the action is done. Finally, it outputs the results.

If you execute this `Function` node, you will get the following output as a result:

```
[
  {
    " myNewField": 1
  }
]
```

So, just how did this work? It worked because of the `items` array, which we are going to be covering in the next section.

The items array

The key to this code is the `items` array. This stores all of the information in the n8n data structure passed to the `Function` node from the previous node. The value in the square brackets represents the index of the JSON object in the `items` array with which you wish to work.

The most basic `items` array only has a single object represented by the zero (0) index number, but it is possible to have several more objects in the `items` array, and you can access each of these objects using the array index number associated with that object.

In the next sections, we are going to talk a bit about referencing different parts of the `items` array using dot notation, along with how to output the information once we have transformed it. Plus, we will also cover how to access data in different nodes other than the one that immediately preceded the node you are in.

Dot notation

Once you have chosen the proper object within the `items` array by indicating its index number, you need to determine if you will work with the JSON or the binary object. You can do this by using what is referred to as **dot notation**. Dot notation allows you to work your way deeper into an array or object by distinguishing parent and child items with a dot or period (`.`). In *line 1* of our code, the dot between `items[0]` and `json` tells n8n to reference the `json` object in the `items` array with the `0` index.

Most of the time, you will be working in the `json` object, but there are situations where you would use the `binary` object.

To better understand the dot notation used to reference different parts of the `items` array, refer to the following table with the dot notation name on the left and the piece of the JSON it is referencing on the right:

Dot Notation	JSON
`items`	`[`
`items[0]`	`{`
`items[0].json`	`"json": {`
`items[0].json.vehicle`	`"vehicle": {`
`items[0].json.vehicle.type`	`"type": "car",`
`items[0].json.vehicle.colour`	`"colour": "red",`
`items[0].json.vehicle.year`	`"year": 2021,`
`items[0].json.vehicle.automatic`	`"automatic": true`
	`}`
	`}`
	`},`
`items[1]`	`{`
`items[1].json`	`"json": {`
`items[1].json.vehicle`	`"vehicle": {`
`items[1].json.vehicle.type`	`"type": "truck",`
`items[1].json.vehicle.colour`	`"colour": "blue",`
`items[1].json.vehicle.year`	`"year": 2009,`
`items[1].json.vehicle.automatic`	`"automatic": false`
	`}`
	`}`
	`}`
	`]`

Outputting data

As with functions in other programming languages, the `Function` node needs to indicate which information gets passed to the rest of the program. This is accomplished by the second line of code, which returns the newly updated `items` array to the next node.

While it is a best practice to modify the value of the `items` array and pass it on to the next node, this is technically not required, and you can return any array that follows the proper n8n data structure.

Data from other nodes (the $items method)

Sometimes, it is necessary to reference the output of a node that is not directly connected to the `Function` node that you are working with but has been executed before your node. In this instance, referencing the `items` array will not give you the correct information.

To resolve this issue, you can use the `$items` method. This method allows you to access all of the data in a node as long as that node is before the present node you are working in and has already been executed.

Let's take a look at this in action. Let's build the following workflow:

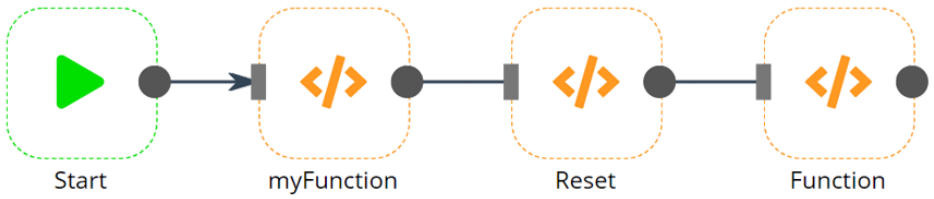

Figure 3.4 – Basic workflow with three Function nodes

Since this is a default `Function` node, the output of the `myFunction` node is this:

```
[
  {
    "myVariable": 0
  },
  {
    "myVariable": 1
  }
]
```

The `Reset` node then deletes everything that it receives and sets the output to empty, as indicated here:

```
[
  {
  }
]
```

We set the JavaScript in the `Function` node to output the information from the previous node, as follows:

```
return items;
```

As you will see, we end up with the same results as we have from the `Reset` node.

Now, let's use the `$items` method to pull the data from the `myFunction` node, essentially skipping the `Reset` node. Change the JavaScript in the `Function` node to this:

```
items = $items("myFunction");
return items;
```

When you run the workflow now, you will see that the output of the `Function` node matches the output of the `myFunction` node.

As with the `items` array, you can specify which item in the `$items` method to reference by setting the index in square brackets. Along with using the array index, you can also use dot notation to reference deeper into the array and objects. For example, this is a perfectly acceptable way of accessing specific information from a node:

```
items[0].json = $items("myFunction")[1].json;
```

Manipulating data

Because the `Function` node uses JavaScript, you have all of the power of JavaScript at your disposal for manipulating data. A deep dive into JavaScript and its capabilities is outside of the scope of this book, but there are some useful commands that we have put together to give you a head start.

Strings

For these examples, we will assume that we have a variable called `fullName`, and the value assigned to it is `Jim Nasium`. Have a look at the following table:

Description	Command	Output
Get the length of a string	`fullName.length;`	10
Connect strings	`"My name is " +fullName;`	My name is Jim Nasium
Find a string in a string	`fullName.indexOf("N");`	4
Extract part of a string	`fullName.slice(4,10);`	Nasium
Replace part of a string	`fullName.replace("J","T");`	Tim Nasium
To uppercase	`fullName.toUpperCase();`	JIM NASIUM
To lowercase	`fullName.toLowerCase();`	jim nasium

Mathematics

JavaScript also has potent mathematical abilities. Here are some practical examples:

Description	Command	Output
Addition	`2 + 3;`	5
Subtraction	`5 - 2;`	3
Multiplication	`2 * 3;`	6
Division	`8 / 2;`	4
Modulus	`10 % 3;`	1
Round down to the nearest integer	`Math.floor(3.6);`	3
Round up to the nearest integer	`Math.ceil(3.6);`	4
Round to the nearest integer	`Math.round(3.6);`	4
Number with the highest value	`Math.max(1,3,5,7);`	7
Number with the lowest value	`Math.min(1,3,5,7);`	1
Return a random number between 0 and 1	`Math.random();`	

As you should now realize, the `Function` node is extremely powerful. We have learned how to reference data both inside the node and from other nodes, manipulate both strings and numbers, and output the data so that it can be used by other nodes.

Now that we have a better understanding of the power of the `Function` node and some ideas of what can be done with the data, let's look at getting some data from remote systems via their API using the `HTTP Request` node.

HTTP Request node – Talk to any API

Computer systems are fantastic for storing and processing massive amounts of data, but computer users and developers very quickly discovered that there was no actual standard for allowing these computers to share information.

Someone had the bright idea to use the new web standard to create a way for these systems to communicate. They would use standard request methods to retrieve, add, delete, or update information on remote computers. Along with these functions, they would even provide the ability to secure these connections using various methods to ensure only authorized individuals could get to the data.

This system setup is collectively referred to as a web API, which is one of the most popular ways of working with remote data today. There are thousands of different APIs available, providing access to an extensive range of data and information.

Since one of the primary functions of n8n is to connect different systems to share data, it only makes sense to talk to these web APIs.

In this section, we are going to learn how APIs work, how they are formatted, ways that data can be passed through APIs, and primary methods supported by APIs, along with the response codes that they return. Finally, we will perform some basic API calls, as well as look at how to secure these calls.

Web API 101

Before we look at the `HTTP Request` node, let's first do a quick overview of how APIs work to better understand how they allow systems to interact with them.

Web APIs run on the same technology as most of the websites on the internet, but instead of dishing up your favorite web pages or cat videos, web servers configured to run APIs allow remote systems to make a request and then reply to the request with data based on the input received earlier. The data that is sent between the systems is often formatted in JSON.

To see the most basic API in action, let's take a look at the Random User API. Open up your web browser, and in the address bar, enter `https://randomuser.me/api/`. When you press the *Enter* key, you will see a bunch of text similar to this:


```
{"results":[{"gender":"female","name":{"title":"Miss",
"first":"Hanna","last":"Farias"},"location":{"street":
{"number":4304,"name":"Rua São Paulo "},"city":"São
Luís","state":"Piauí","country":"Brazil","post-
code":35435,"coordinates":{"latitude":"-77.5289","longi-
tude":"3.6948"},"timezone":{"offset":"-11:00","descrip-
tion":"Midway Island, Samoa"}},"email":"hanna.farias@example.
com","login":{"uuid":"66406cea-46a3-47c4-a9aa-0717ab96ae-
41","username":"redbear531","password":"onlyme","salt":
"xOlBVsMM","md5":"e5aaa5fa1141ced7a3d0b83edbd76ef5","sha1":
"cacfc3f023c50af7c9da3a1bafdd5cd653663ea-
b","sha256":"91e134083ebaf9a721a3f7890be3a186b56a0dde8e-
406f37abf4b68cb41e91a0"},"dob":{"date":"1951-02-27T05:4-
2:48.601Z","age":70},"registered":{"date":"2010-05-19T04:2
1:25.465Z","age":11},"phone":"(65) 4693-1327","cell":"(81)
4446-9285","id":{"name":"","value":null},"pic-
ture":{"large":"https://randomuser.me/api/portraits/women/64.
jpg","medium":"https://randomuser.me/api/portraits/med/wom-
en/64.jpg","thumbnail":"https://randomuser.me/api/portraits/
thumb/women/64.jpg","medium":"https://randomuser.me/api/
portraits/med/women/64.jpg","thumbnail":"https://randomuser.me/
api/portraits/thumb/women/64.jpg"},"nat":"BR"}],"info":
{"seed":"3e10360b4732e141","results":1,"page":1,"ver-
sion":"1.3"}}
```


While this may look like a lot of gibberish, you will see a few characters that look familiar from our `Function` node chapter if you look closely. What we've come across is compressed or unformatted JSON!

Anatomy of an API URL

To access an API, one of the essential items you will need is a **Uniform Resource Locator** (**URL**). It is vital to understand the different parts of a URL because you or your system will often be required to build the URL yourself so that you can work with your desired information.

Let's look at a fictional API URL, dissect the different parts of the URL, and determine their purpose. For this exercise, we are going to use the following URL: `https://api.example.com/v3/computers?type=laptop&ram=32&hdd=1024`

The parts of the API URL are explained in the following sections.

Protocol

In our example, this is the `https://` portion of the URL. This will generally be either `http://` or `https://`. This is important because if the protocol is `http://`, the data is not encrypted between the API server and the client. Anyone can see what information is passing between these two computers, including passwords.

Always make sure that the protocol is `https://`.

Base URL

The base URL in our example is `api.example.com`. It is sometimes referred to as the **hostname**, **domain name**, or **Domain Name System (DNS) name**. These all refer to the same thing. The base URL is generally the server that is hosting the API.

Endpoint

For our example, the endpoint is `/v3/computers`. It is sometimes referred to as the API path. The endpoint is either static (that is, it stays the same) or dynamic (that is, it changes based on the information that is being requested). While there is no absolute standard for how endpoints are used, there are some common practices.

From our example, the `/v3` portion tells us that this is version 3 of the API, and the `/computers` portion tells us that we can expect one or more records to be returned by this endpoint.

Query parameters

The example has three query parameters—`type=laptop`, `ram=32`, and `hdd=1024`. Two delimiters identify the query parameters. The `?` delimiter separates the query parameters from the rest of the URL, and the `&` delimiter separates each query parameter.

Much as with JSON key-value pairs, the key is the portion before the equals sign (`=`), and the value is the portion after the equals sign (`=`).

Other parameters

While using specific endpoints and query parameters are the most common ways of controlling the type of information that is passed between the API client and server, other parameters can further modify the information flow.

Headers

Headers are typically used to provide metadata about the API request. They often give information on the type of data being transmitted, security tokens, or server information.

These are generally transmitted in a key-value pair.

Body parameters

The body of a request typically carries information or data required to complete the request or is supplied by the server based on the request made by the client.

This information is generally sent in a key-value pair.

HTTP methods

There are different ways of interacting with APIs, each producing a different result on the server hosting the API and the client requesting the action. These are referred to as **methods** and can be thought of as action verbs that allow the system to know how to deal with the request. The next sections provide a brief overview of each method and its general use.

GET

`GET` is the most common method. It is used every time you use a web browser to retrieve information from a web server.

`GET` is typically used for retrieving data from an API.

POST

The `POST` method is the other most commonly used method. Web browsers will often use this method to submit information from a web form.

`POST` is generally used for submitting information to an API that is then stored by the API as a new record.

DELETE

The `DELETE` method is usually used to delete a resource or record on the API's remote server.

HEAD

The `HEAD` method works very much like the `GET` method, except that the API only returns header information and no other data.

PATCH

Patching a resource allows you to change only part of the information in the record, leaving everything else the same as it was before.

PUT

`PUT` is similar to the `POST` method in that it creates a new record on the API server if no record exists. But if the information being put to the API has a matching record, this record will be overwritten and replaced by the new information.

Response codes

When a request is made to an API server and the data and metadata is returned to the client, there is also a response code. While diving deep into what each response code means and how you can use them for detailed troubleshooting is beyond the scope of this book, it is essential to know each classification of response code and what they represent.

1xx (informational)

The request was received by the server and is still being processed. Please wait for it to finish.

2xx (success)

Everything worked as expected, more or less.

3xx (redirection)

The API is no longer at the location you requested. Check the error message to get a better idea of the new API location.

4xx (client error)

Something is wrong with how you formed the API request. Check the error message and then update the request.

5xx (server error)

There's a problem with the server, and generally, there is little that you can do to resolve the issue. If you have the contact information for the person/team managing the API server, you could report the error to them and see if they can assist.

Now that we understand what APIs are about, let's start using n8n's `HTTP Request` node to connect to a few APIs.

Basic API call

Let's start with a simple API call. Let's get information from GitHub about the n8n project.

Add a new `HTTP Request` node to the canvas in the n8n Editor UI and open the node. Please leave all of the parameters at their defaults, except for the `URL` value. Set the `URL` value to `https://api.github.com/users/n8n-io/`.

And that's it! We are now ready to retrieve the data from the server via the API! Click on **Execute Node** to perform the API request. You should get a response that looks something like this:


```
[
  {
    "login": "n8n-io",
    "id": 45487711,
    "node_id": "MDEyOk9yZ2FuaXphdGlvbjQ1NDg3NzEx",
    "avatar_url": "https://avatars.
      githubusercontent.com/u/45487711?v=4",
    "gravatar_id": "",
    "url": "https://api.github.com/users/n8n-io",
    "html_url": "https://github.com/n8n-io",
    "followers_url": "https://api.github.com/users/n8n-
      io/followers",
    "following_url": "https://api.github.com/users/n8n-
      io/following{/other_user}",
    "gists_url": "https://api.github.com/users/n8n-
      io/gists{/gist_id}",
    "starred_url": "https://api.github.com/users/n8n-
      io/starred{/owner}{/repo}",
    "subscriptions_url": "https://api.github.com/
```

```
        users/n8n-io/subscriptions",
      "organizations_url": "https://api.github.com/
        users/n8n-io/orgs",
      "repos_url": "https://api.github.com/users/n8n-
        io/repos",
      "events_url": "https://api.github.com/users/n8n-
        io/events{/privacy}",
      "received_events_url": "https://api.github.com/
        users/n8n-io/received_events",
      "type": "Organization",
      "site_admin": false,
      "name": "n8n - Workflow Automation",
      "company": null,
      "blog": "https://n8n.io",
      "location": "Berlin, Germany",
      "email": "hello@n8n.io",
      "hireable": null,
      "bio": "Free and open fair-code licensed node based
        Workflow Automation Tool.",
      "twitter_username": "n8n_io",
      "public_repos": 12,
      "public_gists": 0,
      "followers": 0,
      "following": 0,
      "created_at": "2018-11-30T12:19:59Z",
      "updated_at": "2022-01-07T17:49:22Z"  }
]
```


You can now connect other nodes to the output of the `HTTP Request` node and process the information however you wish, or send it to another system.

Using basic authentication

Now, let's try something a bit more complicated. We're going to look up **Universal Product Code** (**UPC**) values at *UPC Database*. Follow these instructions to set up the user account, get a token for API authentication, and configure a new `HTTP Request` node to query the API:

1. Browse to `https://upcdatabase.org/signup` and create an account.
2. Once you have your account created, go to `https://upcdatabase.org/apikeys` and copy the token on that page. If there is no token on the page, create a new token and then copy it.
3. In the n8n Editor UI, add a new `HTTP Request` node to the canvas.
4. For the **Authentication** parameter, select **Basic Auth**.
5. Next, click **Create New** to open up a new credential window for the **Basic Auth** parameter.
6. Enter in a value for **Credentials Name** (for example, `UPC`), enter your email address in the **User** field, and paste your token into the **Password** field.
7. Click on the **Create** button. This will save the credentials and return you to the `HTTP Request` node. Notice that the **Basic Auth** parameter now has the name of the credentials that you just created.
8. In the URL parameter field, enter `https://api.upcdatabase.org/product/765756931182`.
9. Your `HTTP Request` node should be ready to go. Hit **Execute Node** and check out your results. They should look something like this:


```
[
  {
    "added_time": "2020-04-03 00:28:03",
    "modified_time": "2020-04-03 00:28:03",
    "title": "Raspberry Pi 4 4GB model - New 2019 4GB Ram",
    "alias": "",
    "description": "",
    "brand": "Raspberry Pi",
    "manufacturer": "",
    "mpn": "",
    "msrp": "64.99",
    "ASIN": "",
    "category": "",
```

```
    "metadata": {
      "age": null,
      "size": null,
      "color": null,
      "gender": null
    },
    "stores": null,
    "barcode": "765756931182",
    "success": true,
    "timestamp": 1620901566,
    "images": null
  }
]
```

Try changing the number at the end of the URL and replace it with other UPCs you have around your home, and see what information you can find out about the products.

In this section, we covered the basics of APIs including the parts that make up an API, how information is transmitted using APIs, the different **HyperText Transfer Protocol** (**HTTP**) methods, and their corresponding response codes. We then made both basic and secured API calls.

Now that we better understand APIs, let's move away from retrieving information that we request from other servers and take a look at receiving information that is pushed to n8n.

Webhook node – Handling real-time events

You can think of Webhooks as the cousin of APIs. In fact, you can create an API server using the `Webhook` node!

A Webhook listens for `GET`, `HEAD`, or `POST` requests and then starts a workflow when it detects one. The `Webhook` node can do this because it is a trigger node.

In this section, we are going to learn how to create a Webhook using n8n. As part of this build, we will learn how to send information to the Webhook and how to respond back with the requested information.

Creating a basic test Webhook

To build a basic Webhook, follow these instructions:

1. Add a `Webhook` node to the n8n Editor UI canvas and open the node.
2. Leave all of the parameter values at their defaults.
3. Expand the **Webhook URLs** section by clicking on the **Webhook URLs** text.
4. Click on the **Test URL** toggle under the **Display URL for** section.
5. Click on the URL to copy it to your clipboard (for example, `https://tephlon.app.n8n.cloud/webhook-test/373227bb-5fda-49e9-b491-54ef33db3eed`).
6. Close the `Webhook` node.
7. Save the workflow. This is important because the Webhook URL can't register until the workflow is saved.

That's it! You've just created your first basic Webhook!

Now, let's test it to see if it is working! Follow these steps:

1. In the n8n Editor UI, open up the `Webhook` node.
2. Click on the **Execute Node** button to start the Webhook listening.
3. Open up a new web browser, paste the Webhook URL you copied earlier into the address bar, and press *Enter*.

If everything is working correctly, you should see two things happen, as follows:

1. In the web browser window, you should get the following message:

```
{"message":"Workflow got started."}
```

2. In the n8n Editor UI, you should receive a bunch of information in the `Webhook` node output window, similar to this:

```
[
  {
    "headers": {
      "host": "tephlon.app.n8n.cloud",
      "x-request-id":
        "30d32a403776b5bbce58ce0ef8a4c245",
      "x-real-ip": "10.40.0.4",
      "x-forwarded-for": "10.40.0.4",
```

```
        "x-forwarded-host": "tephlon.app.n8n.cloud",
        "x-forwarded-port": "443",
        "x-forwarded-proto": "https",
        "x-scheme": "https",
        "sec-ch-ua-mobile": "?0",
        "dnt": "1",
        "upgrade-insecure-requests": "1",
        "user-agent": "Mozilla/5.0 (Windows NT 10.0;
          Win64; x64) AppleWebKit/537.36 (KHTML, like
          Gecko) Chrome/90.0.4430.212 Safari/537.36",
        "accept": "text/html,application/xhtml+xml,
          application/xml;q=0.9,image/avif,image/
          webp,image/apng,*/*;q=0.8,application/signed-
          exchange;v=b3;q=0.9",
        "sec-fetch-site": "none",
        "sec-fetch-mode": "navigate",
        "sec-fetch-user": "?1",
        "sec-fetch-dest": "document",
        "accept-encoding": "gzip, deflate, br",
        "accept-language": "en-CA,en-GB;q=0.9,en-
          US;q=0.8,en;q=0.7",
        "cookie":
          "token=eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.
           eyJpc3MiOiJjbG91ZCIsInN1YiI6NDAzMSwi
           dXNlcm5hbWUiOiJ0ZXBobG9uIiwiaWF0IjoxNj
           IwODIwMDYzLCJleHAiOjE2MjE0MjQ4NjN9.
           Q77PkeKY6OUbSJI5Ms56lWvTg8jqSvNOKCp325kTjKo"
      },
      "params": {},
      "query": {},
      "body": {}
    }
  ]
```

This is all of the information that has been sent to n8n from the web browser. While there is some interesting data here, there isn't anything helpful at this time.

Sending information to n8n

Since we have a basic Webhook running, let's see what happens when we send some information to it as a query in the URL. Follow these steps:

1. In the n8n Editor UI, open up the `Webhook` node and press the **Execute Node** button.
2. In the web browser window that you used previously, paste the Webhook URL into the address bar, but before you press *Enter*, add `?fname=Jim&lname=Nasium` to the end of the URL. Then, press *Enter*.

Everything looks the same in the web browser, but let's look closely at the `query` section in the `Webhook` node output window, which you can see here:

```
"query": {
  "fname": "Jim",
  "lname": "Nasium"
},
```

These values match the values entered at the end of the URL that was entered into the web browser. You can now use this information received from the client in the rest of your workflow.

Responding to the client

Now that we know we can receive information from the client, let's send a confirmation message to the sender! Follow these steps:

1. In the `Webhook` node, change `Response Mode` to `Last Node`.
2. Under the **Options** section of the `Webhook` node, add a **Property Name** option and give it a value of `html`.
3. Next, add a **Raw Body** option to the `Webhook` node and enable the option.
4. Close the `Webhook` node.
5. Add a `Set` node to the n8n Editor UI and connect it to the output of the `Webhook` node.
6. Open the `Set` node and add a `string` value.

7. Name the value `html`.
8. Click on the gears next to **Value** and click on the **Add Expression** option.
9. In the **Expression Editor**, paste `<H1>Thanks for visiting, {{$json["query"]["fname"]}} {{$json["query"]["lname"]}}!</H1>` in the **Expression** field.
10. Close the **Expression Editor** and the `Set` node.

Let's see what this does for us. In the n8n Editor UI, click on the **Execute Workflow** button. Then, go back to your web browser and enter the URL you used last time, ending with `?fname=Jim&lname=Nasium`.

If everything has been set up correctly, you should get a message in large letters stating `"Thanks for visiting, Jim Nasium!"`.

How does that work?

When the web browser sends the information to the Webhook, the Webhook grabs the `fname=Jim&lname=Nasium` portion of the URL and sends it on to the `Set` node. The `Set` node then uses the values for `fname` and `lname` to dynamically generate **HyperText Markup Language** (**HTML**), which the Webhook uses to send back to the web browser. The web browser then displays the HTML accordingly.

We've essentially turned n8n into a web server that can generate real-time dynamic web pages!

We have one last section that we want to cover before we close out this chapter. Let's talk about manipulating data stored in arrays and JSON objects.

Working with arrays and JSON objects

n8n uses a lot of arrays and JSON objects, and it is important that you are comfortable working with them. In this section, we will learn how to manipulate arrays and objects by splitting, combining, and writing to these items.

There is a lot of data flying around an n8n workflow, and it can be constructive to learn a few tips and tricks about manipulating this data stored in arrays and JSON objects.

For these examples, we are going to use an array with three JSON objects with the same keys, as illustrated in the following code snippet:

```
[
  {
    "species": "dog",
    "name": "Cocoa",
    "color": "brown"
  },
  {
    "species": "cat",
    "name": "Lucky",
    "color": "brown"
  },
  {
    "species": "cat",
    "name": "Skittles",
    "color": "grey"
  }
]
```

If you look at this information using the **Table** tab at the top of the n8n node generating this information, it will look something like this:

Species	**Name**	**Color**
Dog	Cocoa	brown
Cat	Lucky	brown
Cat	Skittles	grey

You can generate this table in a `Function` node by adding a `Function` node to the n8n Editor UI and then pasting the following into the JavaScript code field:

```
items = [ { "json": { "species": "dog", "name": "Cocoa",
"color": "brown"}}, { "json": { "species": "cat", "name":
"Lucky", "color": "brown"}}, { "json": { "species": "cat",
"name": "Skittles", "color": "grey"} } ];
return items;
```

Now, the output of the `Function` node should match the preceding table.

Next, let's learn how to split data from the `Function` node using the `IF` node.

Separating the cats from the dogs

The first thing we are going to do is split this array into two arrays using the `IF` node. We are going to send all of the dogs to the `true` output and all of the cats to the `false` output.

To do this, follow these steps:

1. Attach an `IF` node to the output of the node generating the array.
2. Open the `IF` node and add a `string` condition to the node.
3. In the `Value 1` field, click on the gears icon and select **Add Expression**.
4. In the **Expression Editor**, enter `{{$json["species"]}}` into the **Expression** field and close the Expression editor.
5. In the `Value 2` field, type `dog` for the value.
6. Close the `IF` node and execute the workflow.
7. Open the `IF` node and take a look at the results for the `true` and `false` outputs. You should see that there are two entries in the `false` output and one in the `true` output.

Combining two arrays

Now that we have split the array apart, let's see if we can bring them back together again. We're going to do this using the `Merge` node. Follow these steps:

1. Add a `Merge` node to the n8n Editor UI.
2. Connect the `true` output from the `IF` node to `Input 1` of the `Merge` node.
3. Connect the `false` output from the `IF` node to `Input 2` of the `Merge` node
4. Open the `Merge` node.
5. For the `Mode` parameter, select **Append**.
6. Close the `Merge` node.

Your final workflow should look something like this:

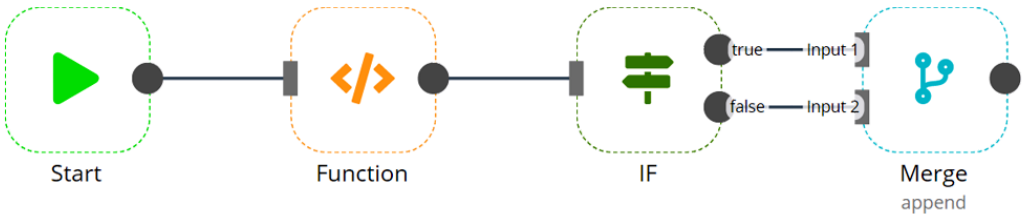

Figure 3.5 – Splitting and merging an array workflow

It's now time to test it out! Execute the workflow. When it finishes running, open up the `Merge` node. The output windows should show that the array is once again back together.

Adding the same value to all JSON objects

Now, let's imagine we want to add another key to all records in the array coming out from the `Merge` node. Let's also assume that each copy of the key will be the same.

We can accomplish this by using the `Set` node. Here are the steps you need to follow:

1. Add a `Set` node to the n8n Editor UI and attach it to the output of the `Merge` node.
2. Open the `Set` node.
3. Add a new Boolean value and name it `adopted`.
4. Set the value of `adopted` to `true`.
5. Close the `Set` node.

To check if it worked, execute the workflow and open up the `IF` node. The table in the output window should now look like this:

Species	**Name**	**Color**	**Adopted**
Dog	Cocoa	Brown	`true`
Cat	Lucky	Brown	`true`
Cat	Skittles	Grey	`true`

Using the `IF`, `Merge`, and `Set` nodes allows us to perform some instrumental data manipulation tasks quickly and easily without resorting to custom coding using the `Function` node.

Summary

This chapter covered some critical concepts, and in it, we learned how to build some powerful tools within n8n.

We first covered how n8n structures data with JSON using the primary components of key-value pairs, objects, and arrays. We followed this by showing how n8n stores JSON and binary data internally. Then we talked about using the `Function` node and understanding the `items` array, dot notation, and the `$items` method. Once functions were figured out, we learned about APIs and how to send and receive data using basic and authenticated calls. Next, we reviewed Webhooks and used them in n8n to send information and generate HTML files. Finally, we went over how n8n works with JSON objects and arrays, including manipulating data that is stored in these items.

With this new information under your belt, you are well on your way to building practical tools using n8n.

In the next chapter, you will put this new knowledge to use and build three applications in n8n!

4
Learn by Doing: Building Two n8n Apps

In this chapter, you will learn to combine concepts from the previous chapters and use them to build multiple projects. Some of these projects will reinforce the concepts you learned earlier and others will introduce some new ideas. This will help you understand the kinds of products that you can build using n8n. Finally, we will learn how to share and discover new workflows as well as participate in n8n's active community.

This chapter will cover the following main topics:

- Building products with n8n
- Building a Telegram bot
- Creating a metrics dashboard
- Sharing and discovering workflows

Technical requirements

This is a list of technical requirements that you'll need to prepare before continuing with the chapter:

- Install n8n.
- Ensure n8n is running and the Editor UI is open.
- Get an account on Telegram.
- Get an account on GitHub.

You can find the completed code examples for the chapter on GitHub at `https://github.com/PacktPublishing/Rapid-Product-Development-with-n8n/tree/main/Chapter%204`

Building products with n8n

If you are anything like me, you have no shortage of ideas for building new and exciting online services or automating boring tasks that are the same every day. Historically, many of these ideas never got any further than that and I didn't know how to get to the next step of actually building the solutions or starting to work on a prototype.

No code tools provide a much simpler way of producing that early **minimum viable product** (**MVP**) or designing the next great web service.

n8n shares this no code philosophy, allowing you to build applications and tools in a fraction of the time that it would take to build them with regular programming languages. Not only is it faster, but it is also easier since all you need to do is understand a bit of JavaScript for the more complicated applications, and maybe not even that, depending on what you want to build!

To get you motivated to start building some applications with n8n, here are a couple of sample apps that perform some fun functions and demonstrate how quick and easy it is to get things up and running.

Building a Telegram bot

Pokémon was one of my favorite shows when growing up. Even today, I enjoy playing *Pokémon Go*. At times, you will come across very tough opponents in the game and you have to do some research on what the weakness of the opposing Pokemon is. My search history is full of queries like, "How to defeat Tyranitar." Let's create a Telegram bot that gives us information about a Pokemon's abilities, moves, and types.

In *Chapter 2*, *Using the Editor UI to Create Workflows*, we learned how to create Telegram bots, as well as how to query a REST API. We'll be using a similar workflow but with some catches:

- We'll have to check whether the user of the bot has entered the name of a Pokemon along with the slash command.
- We'll query the Pokemon API (`https://pokeapi.co`) to get the data. However, that API is very comprehensive and we'll need to massage the data so that it fits the format that we want to consume this information in.

Let's get started by creating a new Telegram bot by following the instructions provided in *Chapter 2, Using the Editor UI to Create Workflows*. I named my bot **Pokemon Bot**. Next, we'll need to set a command for our bot. Go to the chat with **BotFather** and enter `/setcommands`. Choose the bot that you just created, and enter the following:

```
pokemon - Get details about a specific pokemon
```

You should be able to see something similar to the following screenshot after running the command:

Figure 4.1 – Setting a command for our Telegram bot

Congratulations, you have just created the first command for your bot.

Let's start building the backend for this bot in n8n. To do this, follow these steps:

1. Open your n8n Editor UI and add a **Telegram Trigger** node.
2. Enter the credentials for your new bot and select * from the drop-down list for the **Updates** field.
3. Now, save and execute the workflow.

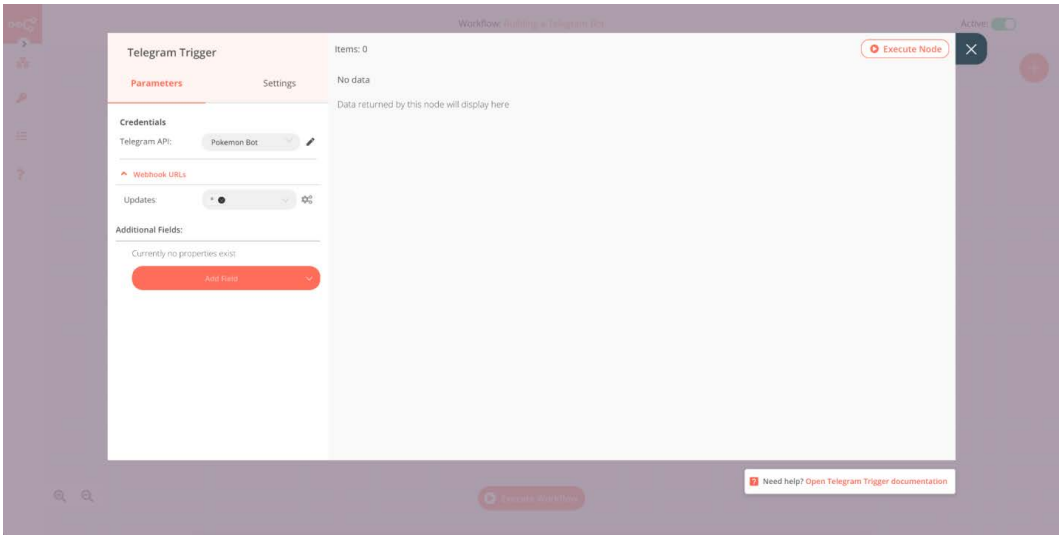

Figure 4.2 – Settings for the Telegram Trigger node

> **Note**
>
> If you are running n8n locally, make sure that you are running using tunnel mode.

4. Now, go to your bot and enter `/pokemon ditto`.

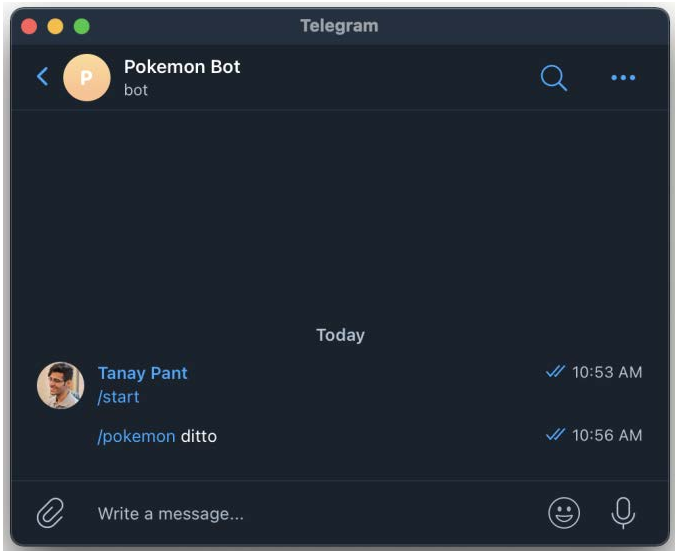

Figure 4.3 – Sending a command to the Pokemon bot

5. Go back to the Editor UI and you will see a response there. It is likely that the response is `/start`. If it is, press **Execute Workflow** again, until you see `/pokemon ditto` in the respons

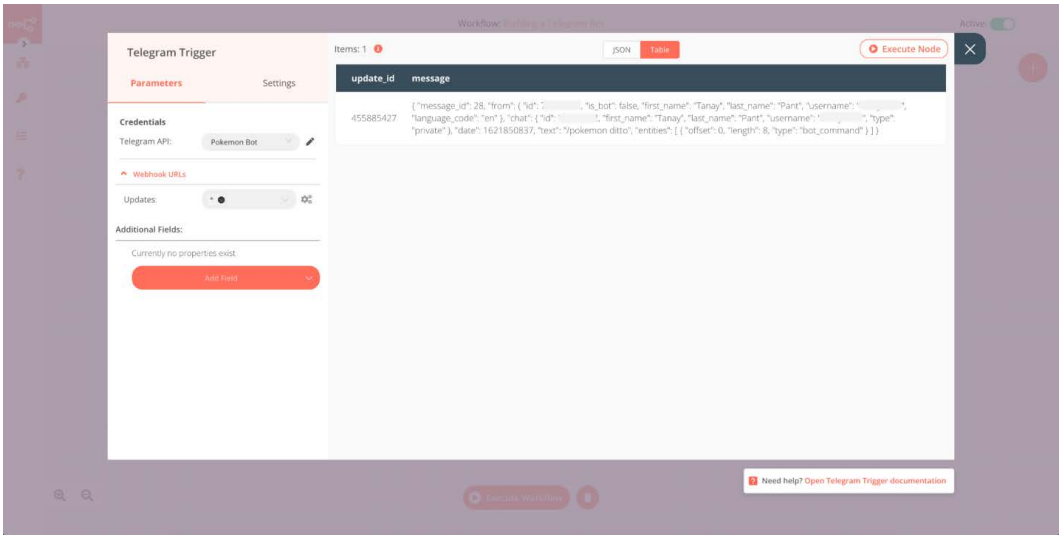

Figure 4.4 – Receiving /pokemon ditto as a command with the Telegram Trigger node

You will notice that it is quite likely that someone just clicks on the command without entering the name of the Pokemon. We will have to account for the fact that someone might make the same mistake while using this bot. To make sure that this does not happen, let's add an **IF** node to check whether the user provided the name of a Pokemon.

Add an **IF** node to the Editor UI and make sure that it is connected to the **Telegram Trigger** node. Enter the Node Editor view, click on the **Add Condition** button, and select **String**. Select **Is Empty** for **Operation**. Add an expression to the **Value 1** field and enter the following:

```
{{ $json["message"]["text"].split(' ')[1] }}
```

This JavaScript snippet points to `'/pokemon ditto'` using `$json["message"]["text"]`. The `.split(' ')` method splits the string at the space and converts it into an array, which looks like this: `['/pokemon', 'ditto']`. Then we select the item at position 1, which is `'ditto'`. Execute the node.

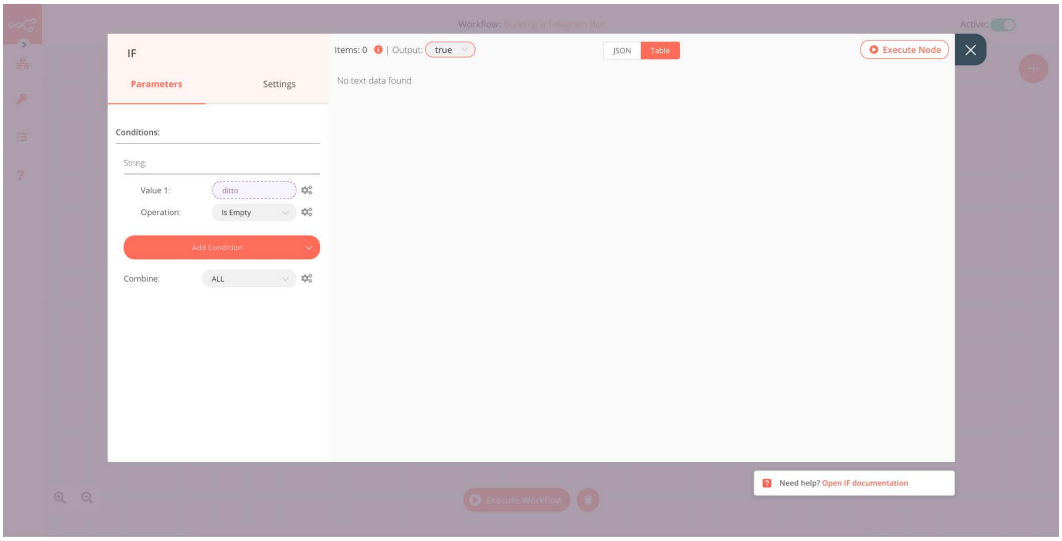

Figure 4.5 – Executed IF node with no output for true

You will notice that the output for the **true** section is empty since the condition that we specified is false. If you select **false** instead of **true** for the **Output** field at the top, you will see one result. Let's now set a default message in case the name of the Pokemon has not been specified.

1. Add a **Telegram** node to the Editor UI and connect it to the true output of the **IF** node.
2. Configure the credentials for the node, and use the expressions to set the value for **Chat ID** (you can get it from the **Telegram Trigger** node). The expression should look something like this:

```
{{$node["Telegram1 Trigger"].json["message"]["chat"]
["id"]}}
```

> **Note**
>
> We used an expression here as compared to the **Telegram** node in *Chapter 2, Using the Editor UI to Create Workflows,* since many people might be using the **Telegram** bot and we want to send the answer to the person who queried it.

3. Enter the following in the text field, 👀 `Please enter the name of a Pokemon. For example, '/pokemon ditto' (without the quotes)`. Feel free to customize the message.
4. Now let's execute the workflow and this time, let's just send `/pokemon` to the bot. The response should look something like this.

Figure 4.6 – Response after sending /pokemon as the command to the bot

Now that we have cleared one of the challenges that we mentioned before, let's focus on getting the data from the API and sending it back to the user. To do this, follow these steps:

1. Add an **HTTP Request** node to the Editor UI and connect it with the **false** output of the **IF** node. The workflow should now look like this.

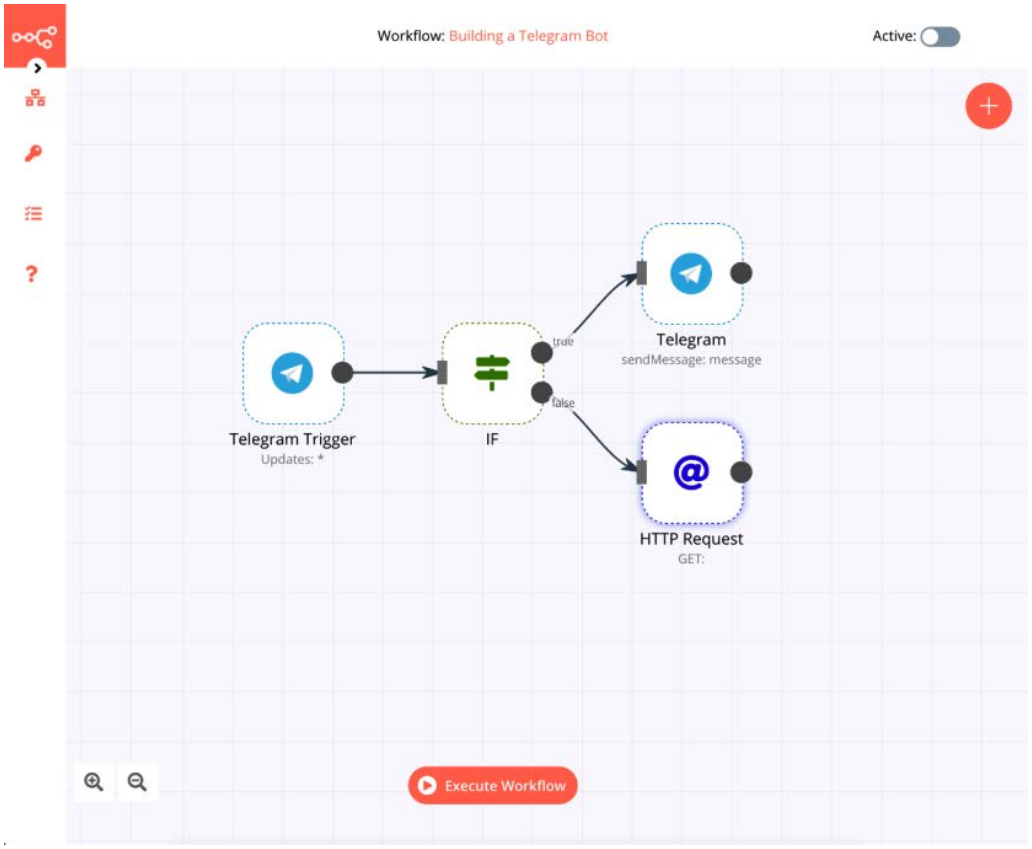

Figure 4.7 – Your workflow should look like this after adding the HTTP Request node

Before we move forward, execute the workflow again, and send `/pokemon ditto` to the bot. This will make the data in the workflow flow toward the false branch as we build that branch.

2. Open the **HTTP Request** node, add an expression to the **URL** field, and enter the following:

```
https://pokeapi.co/api/v2/pokemon/{{$json["message"]
["text"].split(' ')[1]}}
```

This will ensure that the API returns details about the Pokemon that you asked for. Execute the node and you will notice that you get loads of data about Ditto. We are specifically interested in the abilities, moves, and types. However, these are arrays with objects in them. We'll have to massage the data so that it converts into a format that is useful for us. We will use the **Function** node to do that.

Add a **Function** node to the Editor UI and connect it to the **HTTP Request** node. Open the **JavaScript Code** field and enter the following code:

```
const abilities = [];
const moves = [];
const types = [];

for (let i=0; i<items[0].json.abilities.length;i++) {
abilities.push(items[0].json.abilities[i].ability.name);
}

for (let i=0; i<items[0].json.moves.length;i++) {
moves.push(items[0].json.moves[i].move.name);
}

for (let i=0; i<items[0].json.types.length;i++) {
types.push(items[0].json.types[i].type.name);
}

return [{json: {name: items[0].json.name, abilities, moves,
types}}];
```

Let's understand what is happening here. We created three new arrays called `abilities`, `moves`, and `types`. We then created a loop that will push the name of each ability, move, and type into its respective array. Finally, we returned the data in a format expected by n8n. Execute the node and it should then look like this:

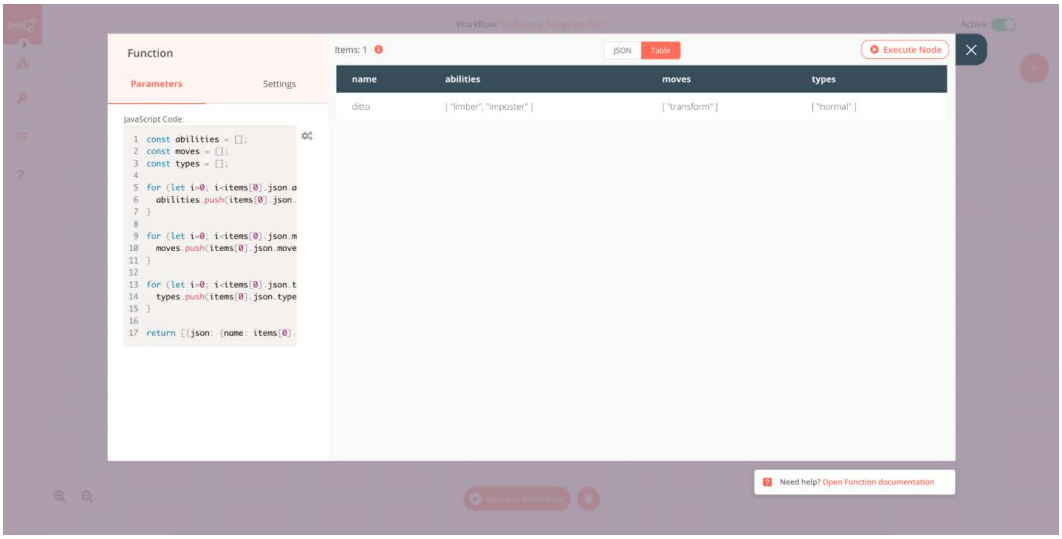

Figure 4.8 – Output of the Function node

Perfect, we now have just the data that we need. Now we need to send it to the Telegram bot.

Add a new Telegram node and connect it with the **Function** node. Configure your credentials and **Chat ID** as discussed before. Now, add an expression to the **Text** field and format the data as you see fit.

Here's what my expression looks like:

```
<b>Name:</b> {{$json["name"]}}
<b>Abilities:</b> {{$json["abilities"].join(', ')}}
<b>Moves:</b> {{$json["moves"].join(', ')}}
<b>Types:</b> {{$json["types"].join(', ')}}
```

Notes

When adding values to expressions that are an array, such as `abilities`, you can click on the gray dot next to the value in the expression editor, and click on **Values** as shown in the following screenshot.

In the preceding example, I have used the bold HTML tags to bold some text. If you would like to include that as well, click on the **Add Field** button, select **Parse Mode**, and set it to **HTML**.

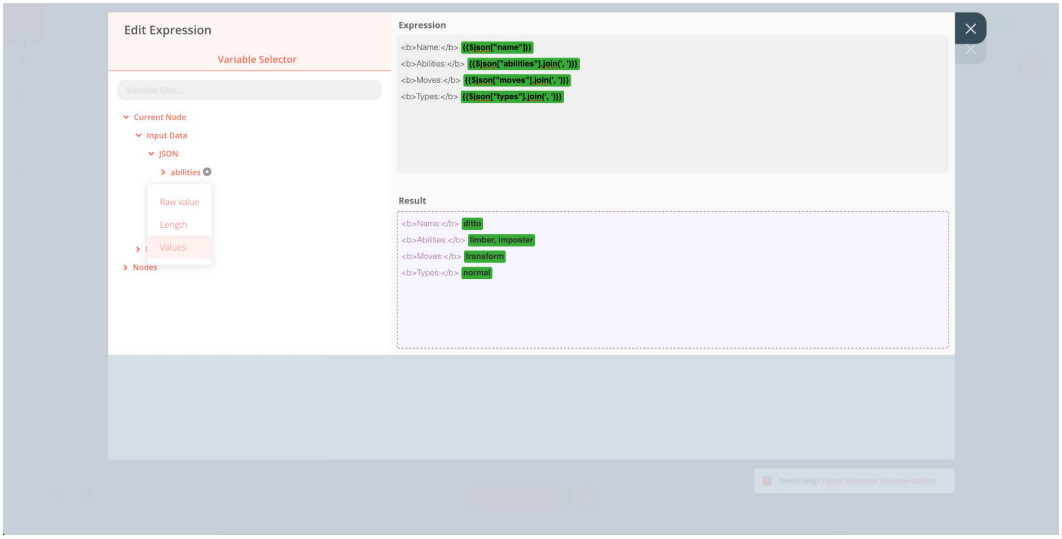

Figure 4.9 – Adding values from an array in the expression editor

Now, save and activate your workflow. Here's what your workflow should look like:

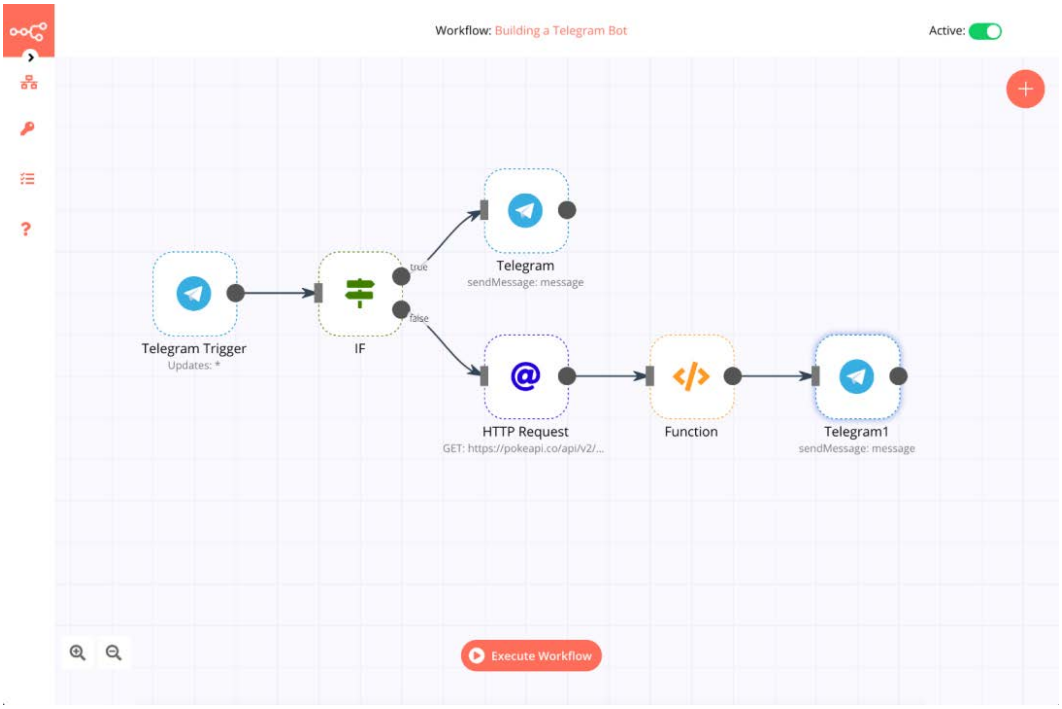

Figure 4.10 – Final workflow

Now, go ahead and enter `/pokemon ditto` in your Telegram bot. Here's what the result looked like for me:

Figure 4.11 – Result of the /pokemon ditto command in the Telegram bot

Go ahead and try a few more. Here are some names of Pokemons: Meowth, Pikachu, Bulbasaur. Gotta catch them all.

Let's use our knowledge of webhooks from *Chapter 3, Diving into Core Nodes and Data in n8n*, to build a metrics dashboard with n8n.

Building a metrics dashboard

Metrics are a key component of any business. It's important to keep an eye on metrics to measure the health and growth of communities, products, revenue, and many other things. Let's build a metrics dashboard that will show us the count for the following:

- GitHub stars
- GitHub forks
- Docker pulls

It should be straightforward to add or redact any other numbers that you'd like to see in there. There are two main components to building this numbers dashboard:

- Serving the web page for displaying the metrics in an easy-to-read format
- Getting the data from different services and inserting it into the web page

Let's start off by learning how to serve a web page with n8n. The following are the steps for it:

1. Open your Editor UI and add a **Webhook** node. Select **Last Node** from the drop-down list for the **Response Mode** field.
2. Click on the **Add Option** button and select **Property Name**. Enter `html` into the **Property Name** field.
3. Save the workflow and execute the workflow. Copy the test webhook URL and paste it into your web browser. The **Webhook** node should look something like this now.

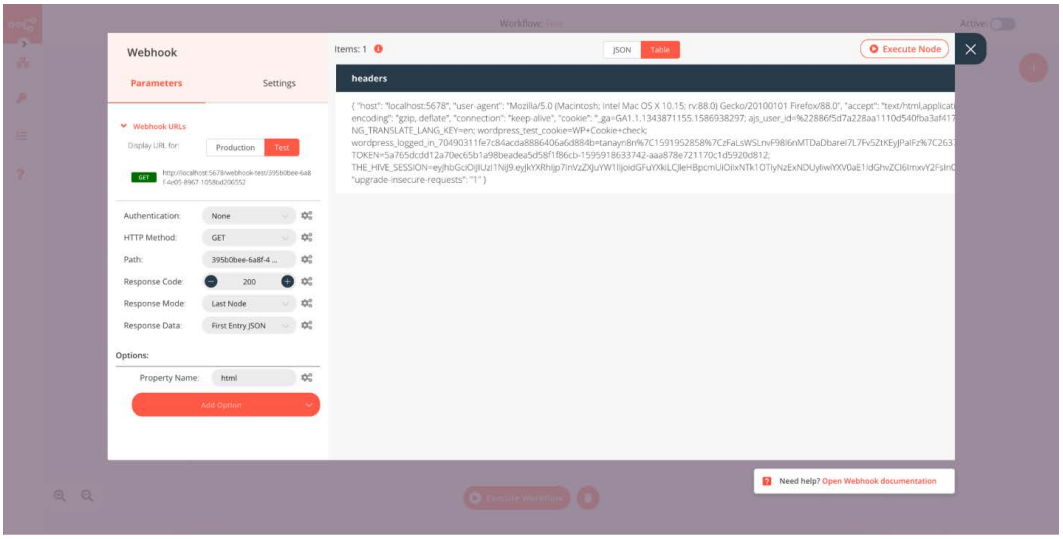

Figure 4.12 – Configuring the Webhook node to serve web pages

4. Add a **Set** node to the Editor UI and connect it to the **Webhook** node. Toggle the **Keep Only Set** button to true (green).

5. Click on the **Add Value** button and select **String**. Enter `html` in the **Name** field and enter the following expression in the **Value** field:

```
<html>
<body>
<h1>From n8n with love 🖐</h1>
<b>Host:</b> {{$json["headers"]["host"]}}
</br>
<b>User Agent:</b> {{$json["headers"]["user-agent"]}}
</body>
</html>
```

Here, we have added some HTML and used the expressions to point to the host and user-agent values that the **Webhook** node provided. Execute your workflow and open the test webhook URL again in your browser. You should see something like this.

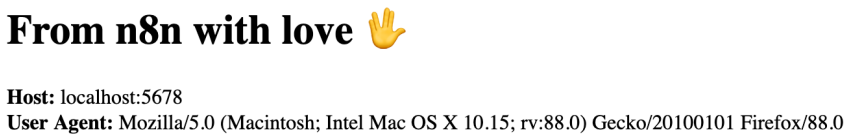

Figure 4.13 – Web page served by the n8n workflow

Now that we know how to serve web pages with n8n, let's take a look at the next piece of the puzzle – getting the data from GitHub and Docker Hub. To do this, follow these steps:

1. In your current workflow, delete the **Set** node. Add the **GitHub** node to the Editor UI and connect it with the **Webhook** node.
2. Configure your credentials for the **GitHub** node by following the steps given here: `https://docs.n8n.io/credentials/github`.

3. Select **Repository** for **Resource** and **Get** for **Operation**. Fill in the **Repository Owner** and the **Repository Name** fields. I filled `n8n-io` and `n8n` in the fields respectively.
4. Click on the **Execute Node** button and it should return details about the repository that you specified. It should look something like this:

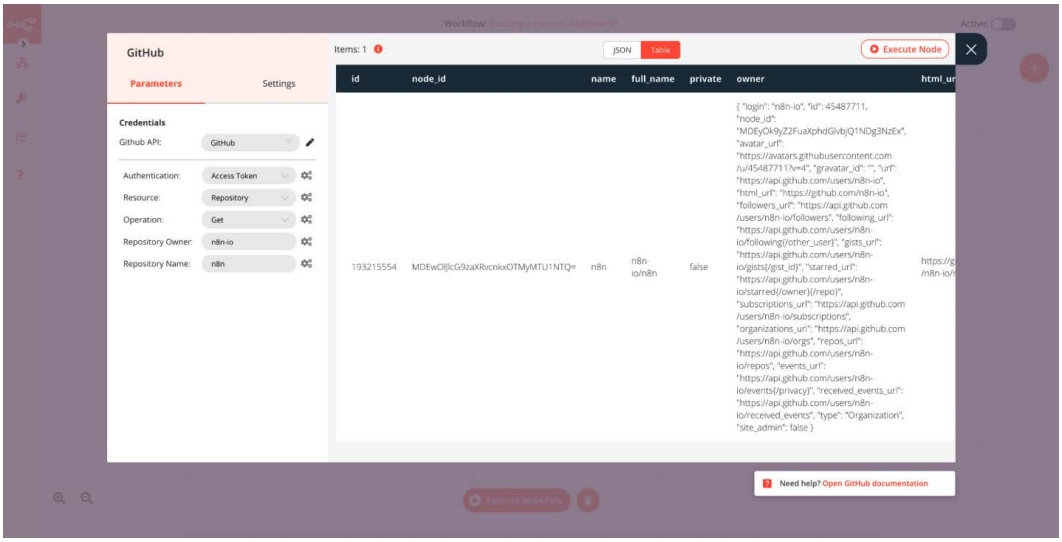

Figure 4.14 – Output from the GitHub node

If you browse through the output from this node, you will notice that we get the star and fork counts as a result too. Next, we need to get the number of pulls from Docker Hub. We'll use the **HTTP Request** node to do that.

5. Add the **HTTP Request** node to the Editor UI and connect it to the **GitHub** node. Enter the following URL in the **HTTP Request** node: `https://hub.docker.com/v2/repositories/n8nio/n8n`. Feel free to replace this with the URL of your repository. Execute the node and the output should look like this:

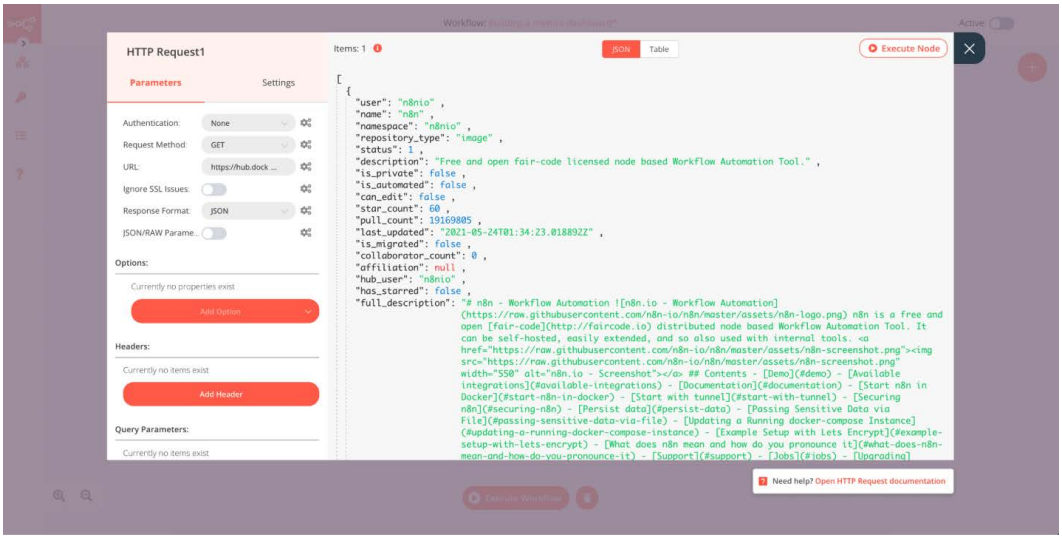

Figure 4.15 – Output of the HTTP Request node

Now that we have all the information that we need, we just need to create a nice HTML template to display all this information and we will be ready.

6. Add a **Set** node to the Editor UI and connect it to the **HTTP Request** node. Toggle the **Keep Only Set** field to true (green).

7. Click on the **Add Value** button and select **String**. Enter `html` in the **Name** field and add an expression to the **Value** field. Paste this HTML code in the expression editor:

```
<html>
<head>

Add styling to our dashboard so that it looks nice

<style>
@importurl(https://fonts.googleapis.com/
css?family=Droid+Sans);
@importurl(http://weloveiconfonts.com/
api/?family=fontawesome);
/* fontawesome */
```

```
[class*="fontawesome-"]:before {
font-family: 'FontAwesome', sans-serif;
}
* {
     margin: 0;
     padding: 0;
     border: 0;
     font-size: 100%;
     font: inherit;
     vertical-align: baseline;
     -webkit-box-sizing: border-box;
     -moz-box-sizing: border-box;
     box-sizing: border-box;
}
.fl{ float:left; }
.fr{ float: right; }
/*its also known as clearfix*/
.group:before,
.group:after {
     content: "";
     display: table;
}
.group:after {
     clear: both;
}
.group {
     zoom: 1;  /*For IE 6/7 (trigger hasLayout) */
}

body {
     background: #F2F2F2;
     font-family: 'Droid Sans', sans-serif;
     line-height: 1;
     font-size: 16px;
}
.pricing-table {
```

```
        width: 80%;
        margin: 50px auto;
        text-align: center;
        padding: 10px;
        padding-right: 0;
    }
    .pricing-table.heading{
        color: #9C9E9F;
        text-transform: uppercase;
        font-size: 1.3rem;
        margin-bottom: 4rem;
    }
    .block{
        width: 30%;
        margin: 015px;
        overflow: hidden;
        -webkit-border-radius: 5px;
        -moz-border-radius: 5px;
        border-radius: 5px;
    /*     border: 1px solid red;*/
    }
    /*Shared properties*/
    .title,.pt-footer{
        color: #FEFEFE;
        text-transform: capitalize;
        line-height: 2.5;
        position: relative;
    }
    .content{
        position: relative;
        color: #FEFEFE;
        padding: 20px010px0;
    }
    .price{
        position: relative;
        display: inline-block;
```

```
      margin-bottom: 0.625rem;
  }
  .pricespan{
      font-size: 3rem;
      letter-spacing: 8px;
      font-weight: bold;
  }
  .pt-footer{
      font-size: 0.95rem;
      text-transform: capitalize;
  }
  /*PERSONAL*/
  .block.personal.fl{
     background: #78CFBF;
  }
  .block.personal.fl .content,.block.personal.fl
  .pt-footer{
      background: #82DACA;
  }
  .block.personal.fl .content:after{
  border-top-color: #82DACA;
  }
  .block.personal.fl .pt-footer:after{
      border-top-color: #FFFFFF;
  }
  .block.business .title{
      background: #3EC6E0;
  }
  .block.business .content,.professional .pt-footer{
      background: #53CFE9;
  }
  .block.business .content:after{
  border-top-color: #53CFE9;
  }
  .block.business .pt-footer:after{
      border-top-color: #FFFFFF;
```

```
}
/*BUSINESS*/
.block.business.fl .title{
     background: #E3536C;
}
.block.business.fl .content, .block.business.fl
.pt-footer{
     background: #EB6379;
}
.block.business.fl .content:after{
border-top-color: #EB6379;
}
.block.business.fl .pt-footer:after {
border-top-color: #FFFFFF;
}
</style>
</head>

Create three different div elements to position and
display the three metrics

<body>
     <div class="wrapper">
     <div class="pricing-table group">
     <h1 class="heading">
     Metrics overview
     </h1>
     <div class="block personal fl">
     <h2 class="title">GitHub Stars</h2>
     <div class="content">
     <p class="price">
     <span>{{$node["GitHub"].json
        ["stargazers_count"]}}</span>
     </p>
     </div>
     </div>
     <div class="block business fl">
```

```
        <h2 class="title">GitHub Forks</h2>
        <div class="content">
        <p class="price">
        <span>{{$node["GitHub"].json["forks"]}}</span>
        </p>
        </div>
        </div>
        <div class="block business">
        <h2 class="title">Docker Pulls</h2>
        <div class="content">
        <p class="price">
        <span>{{$node["HTTP
          Request"].json["pull_count"]}}</span>
        </p>
        </div>
        </div>
        </div>
        </div>
    </body>
    </html>
```

We have used some HTML and CSS to make the metrics dashboard presentable. You will notice that we have three different containers for the three different values that we are pulling from GitHub and Docker Hub.

8. Save and activate the workflow. Now grab the production webhook URL and open it in your browser. It should look something like this:

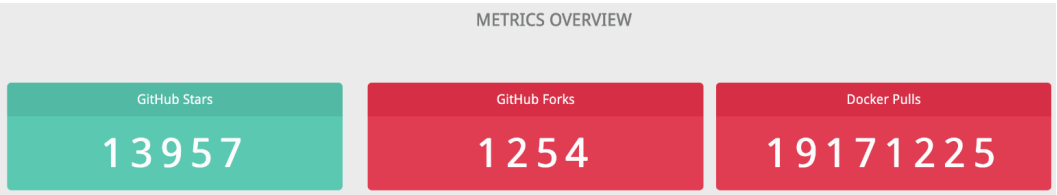

Figure 4.16 – Opening the dashboard using the production webhook URL

Congratulations, you have built a metrics dashboard that pulls in data from two different sources and displays it using a web page that is served from your n8n workflow.

As we move forward, you'll be building a lot of different workflows. Let's take a look at how you can discover and share workflows with the n8n community.

Sharing and discovering workflows

Sometimes, it makes sense to share your workflows with the community to showcase what you have been building and inspire them. It is also useful to discover workflows submitted by the other community members to get inspiration for your next project.

The n8n.io website has a page dedicated to sharing and discovering workflows. You can access the page here: `https://n8n.io/workflows`. You can submit workflows with a title, description, and JSON. This is what a submitted workflow looks like:

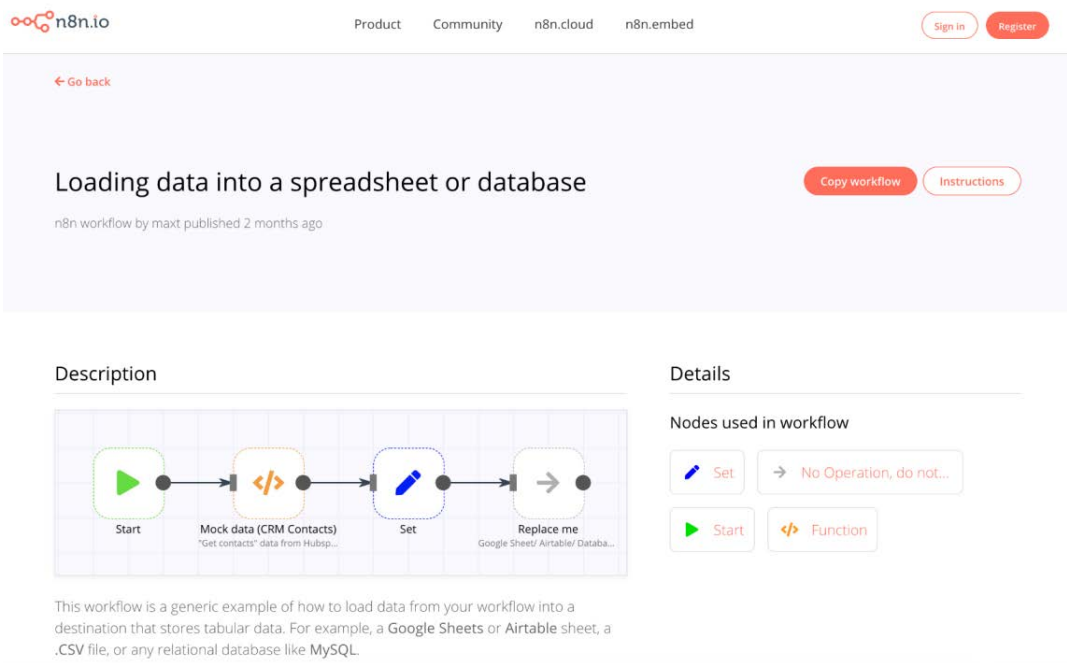

Figure 4.17 – Example of a submitted workflow on n8n.io/workflows

The website automatically recognizes the nodes that you have included in the workflow and lists them on the right side. When submitting a workflow, make sure of the following:

- The specific workflow doesn't already exist.
- The title is descriptive.
- The description has an explanation of what the workflow does and a screenshot of it to aid the explanation.

This is also a great place to discover workflows submitted by other members of the community to gain inspiration. While building your workflows, if you run into any trouble, you can always post about your problem in the community forum: `https://community.n8n.io/`.

n8n has a very active community and the forum is very useful for getting timely help whenever you are stuck. We, ourselves, also spend a lot of time answering questions on the forum, so if you get stuck at any point during the book, tell us about it in the community forum and we'll be there to help you out. Our usernames in the forum are **@tanay** and **@tephlon**.

Summary

In this chapter, we learned about why it makes sense to build products with n8n and translated two of our ideas for products into n8n workflows by building a Telegram bot and a **metrics dashboard**. Having a product mindset and an understanding of which nodes to use to move your project forward will be useful in the next chapters, where we will build our own project with n8n and Bubble.

In the next chapter, we will examine how to build a common way of communicating between modern systems: the **application programming interface**, or **API**, as it is more commonly known. We will learn how to use n8n to build APIs for both exposing and collecting data for systems that do not have this way of accessing information already.

Section 2 – Building an API to Power Your Application

In this section, you will build a complete API solution for receiving and responding to real-time requests from external systems.

In this section, there are the following chapters:

- *Chapter 5*, *Building Your First API Endpoints*
- *Chapter 6*, *Powering Your API with a No Code Database*
- *Chapter 7*, *Transforming Your Data inside a Workflow*
- *Chapter 8*, *Utilizing the Bubble API in n8n*

5
Building Your First API Endpoints

Communication is at the core of all relationships. Good or bad relationships are often defined by how well two individuals can convey their thoughts, feelings, and ideas. Communication needs to be clear, accurate, and trusted.

Sharing information appropriately is also critical for computer applications. Having a standard way of sending and receiving data between two different applications is critical for many programs.

One of these standards is referred to as an **API**, which stands for **application programming interface**. We are working specifically with the REST API architecture style. It was designed to quickly provide information and perform actions based on standard HTTP methods.

Because of this well-known and understood standard, in this chapter, we will show you how to plan and build an API using n8n. We will help you plan your API by understanding the core concepts of APIs and thinking about what you want the final product to perform. You will also understand how to use n8n to build, secure, and test the API that you designed.

In this chapter, we are going to cover the following topics:

- Planning your project's API
- Configuring the Webhook node to handle requests
- Building the API in n8n
- Securing your API endpoints
- Testing your API

By the end of the chapter, you will be able to do the following:

- Create blueprints for an API endpoint so that there's minimal friction when building the API.
- Configure the Webhook node so that it can handle requests that are sent to our API and reply to them.
- Build an API endpoint in n8n based on the blueprints that we created.
- Secure your API endpoints by using the different authentication methods available in the Webhook node.
- Test your API to make sure that all the functionalities that we have implemented work as expected.

Technical requirements

The following are the technical requirements for this chapter:

- A working version of n8n.
- A web browser.
- Access to this book's GitHub repository: `https://github.com/PacktPublishing/Rapid-Product-Development-with-n8n`. Another useful tool is the Insomnia API tool, which can be found at `https://insomnia.rest/`.

Planning your project's API

Before you start building out your API, you must have a solid roadmap. This plan allows you to create the API quicker and ensure the design is consistent and accurate.

There are several different dimensions to consider when you are planning out your API. For an example of good API design, see the OpenAPI Specification at `https://swagger.io/specification/`.

Easy to understand

How an API works should be almost obvious to a developer using it. The API should have terms that are consistent with similar APIs and adequately describe the information it is using and the action it is performing.

Output data in JSON

While several different data formats are available to use, such as XML, YAML, and SOAP, REST APIs most often output their data in JSON format. To maintain consistency, it is recommended that all of your APIs, at the very least, be able to output their data in JSON.

Using the GET, HEAD, and POST HTTP methods

n8n supports three different HTTP methods (you can think of HTTP methods like action verbs that tell the system what to do with the request it has received):

1. `GET`
2. `HEAD`
3. `POST`
4. `DELETE`
5. `PATCH`
6. `PUT`

While several other HTTP methods are available, such as `UPDATE`, at the time of writing this book, n8n does not support them.

The two most commonly used HTTP methods are `GET` and `POST`, each of which provides a specific action.

The `GET` method is generally used when we're attempting to retrieve (or get) information from the API. It should not be used to pass sensitive information to the API. It is common for the query parameters to be passed along in a URL format that is human readable and often stored in web browser history.

However, the flip side of this method is that it is straightforward to create a website URL that specifies all the information being requested by the user in a simple link.

> **Note**
> This assumes that no security has been applied to the API that needs to be passed in the headers.

The `POST` HTTP method is usually used to create or add a new record to the system providing the API. In n8n, it is also often used to change a record since there is no `UPDATE` HTTP method support.

One of the advantages of using the `POST` HTTP method is that it does not show up in the URL when it is submitted to the server.

> **Note**
> For more information about response codes, check out the *HTTP methods* subsection of the *HTTP Request node – talk to any API* section of *Chapter 3, Diving into Core Nodes and Data in n8n*.

Knowing what your API will do

It is critical to know precisely what your API is meant to do, not only in terms of the capabilities of the API but also inside the system. A simple API call can efficiently perform many different actions behind the scenes.

Plan out what each API call does in detail once the system receives the API request. This should be very detailed and documented exceptionally well.

Having meaningful and consistent response codes

Several different response codes can be used in response to an API call. Still, it is generally recommended that you limit yourself to the `2xx` and `4xx` codes since most other codes are handled by other systems, such as the web server (although we have an exception for this).

In general, it is recommended that you use the following code for each stated purpose:

- `200 (OK)`: All the data has been received, processed, and returned correctly. No further actions are required.

- `201 (Created)`: Often used with `POST` requests, this indicates that the request has resulted in creating a new resource and that the information for the resource has been returned in the results.
- `202 (Accepted)`: When information is processed asynchronously, the system may have been received but is not yet active in the system, so the API could return a `202` code. This lets the requester know that the information made it to its destination, but that it cannot be accessed yet.
- `203 (Non-Authoritative Information)`: If you are caching local data to reduce the number of calls to a remote data source, your API can respond with a `203` response code to let the user know that the information they received was accurate the last time it pulled a copy of the information, though this could have changed in the interim.
- `204 (No Content)`: Sometimes, a request for information is made to an API, but there is nothing to return to the user. For example, if an API references a list of animals (for example, cat, dog, and horse) and the user has requested a list of all reptiles, there would be no animals in the results. This is when the API would respond with a `204` response code so that the user knows that the API meant to send back no information and that there wasn't some error returning no results from the API.
- `206 (Partial Content)`: It is not uncommon for an API `GET` request to generate several thousands of records. Returning all of these records can be very taxing on both the server hosting the API and the client receiving the data. There can also be a considerable amount of bandwidth used when transmitting large amounts of data.
- The API only sends out a portion of the records (often referred to as pagination) to the requestor to alleviate this issue. It includes a `206` response code to let us know that there is more information to be received.
- `400 (Bad Request)`: It is straightforward to miswrite an API request, and it is important to let the requester know that they made a mistake. For example, if a requester was searching for a user by email address but asked the system to look for the information in the first name field, then a `400` response would be warranted.
- `401 (Unauthorized)`: You should secure your API with some sort of authentication scheme. When someone provides the wrong authentication information, a `401` response code should be generated and returned to the user.

- `403 (Forbidden)`: Sometimes, people attempt to access information that they shouldn't, regardless of whether or not they are properly authenticated to the system. In this instance, a `403` response code would be returned to the user. I often use this response code when I have my API set up to only be accessed from a specific IP address and the IP addresses do not match.
- `404` (`Not Found`): It is not uncommon for users to accidentally enter the wrong information for a URL or for a resource to move. When that happens, the user making the request will automatically get a 404 error.

> **Note**
>
> For more information about response codes, see the *Response codes* section of *Chapter 3*, *Diving into Core Nodes and Data in n8n*.

Consistent noun/verb design

One very common way of laying out an API is by using a **noun/verb** architecture. The idea behind this design is that the requestor starts with an object or item that is directly followed by an action to be performed in that object.

For example, if we were to create an API that would change the name of a device in its database, the request path may look something like this:

```
/device/rename
```

Similarly, if we were to remove that device from the database, the API call path might look something like this:

```
/device/remove
```

Submitting data

There are several different ways to submit data to an API. It is entirely up to you how you want your users to send information to your API:

- **Body**: The body of the API request is a common place to place information. This is most often used in POST HTTP requests.
- **API Path**: The API path is a different way of sending information. It is often used when you're referencing a specific record item and then performing an action on that item. For example, if you needed to delete record ID 237 from the system, you may have the user send a request to `/record/237/delete` to accomplish this task.

- **Query**: A query is often used with the `GET` HTTP method as it is used to ask for specific information. A query is often represented by a URL and is even displayed that way. For example, if you were to request record 237 from the system using a query, the URL path may look something like `https://api.example.com/record/display?recordID=237`. The `?` character separates the API path and the query. `recordID` is the data key to be searched for, while `237` is the value of the key the system is looking for. You can also have multiple key/value pairs in a query by concatenating the queries together with a `&` character; for example, `recordID=237&fname=Tim`.
- **Header**: You can also send information as a part of the headers in the request. This is often done when you're sending authentication information such as an API token to the API.

Versioning your API

It is not uncommon that, as your system changes and matures, you may want to make changes to your API. The problem with modifying how your API works is that the changes can suddenly make a large number of user programs and scripts fail. This quickly leads to several upset and disgruntled customers.

One easy way around this is to create multiple versions of your API. This way, people can continue to use their scripts with previous versions of the API while new users can automatically move to the new version.

However, there is an issue with creating multiple versions of your API. As the number of versions grows, so does the effort required to maintain the system and all of its versions.

To avoid this ever-growing effort, it is recommended that you maintain no more than two versions of an API at any given time. When a new version of the API is released, clearly communicate to the users that the previous version has been replaced and that the old version will be decommissioned in a reasonable amount of time ("**reasonable**" depends on the audience, where it could be a couple of weeks, or several years).

Documenting your API

One of the most important parts of the design process is documenting the API so that others can use it properly. This can also help you figure out what you did when you come to troubleshoot it 2 years later.

The OpenAPI Specification

One of the more common forms of documentation is the OpenAPI Specification (`https://github.com/OAI/OpenAPI-Specification`). This specification only takes a couple of hours to learn and can be used in conjunction with other tools to automatically create an API testing platform and all user documentation.

The specification uses either a JSON or YAML file to outline how your API behaves and how to create the documentation for the specification.

The following is a simple JSON OpenAPI Specification file's contents:

```
{"openapi": "3.0.0",
  "servers": [
    {"description": "User Example",
      "url": "https://virtserver.swaggerhub.com/
        tephlon/user-example/1.0.0"}
  ],
  "info": {"version": "1.0.0",
    "title": "User Example",
    "description": "Add a user with the API"},
  "paths": {
    "/api/v1/user/add": {
      "post": {
        "tags": ["Users"],
        "operationId": "addUser",
        "parameters": [
          {"name": "email",
            "in": "query",
            "description": "User's Email Address",
            "schema": {"type": "string"}
          },
          {"name": "password",
            "in": "query",
            "description": "User's Password",
            "schema": {"type": "string"}
          }
        ],
        "responses": {
```

```
            "200": {
              "description": "User Added Successfully",
              "content": {
                "application/json": {
                  "schema": {
                    "properties": {
                      "userID": {"type" : "integer"}
                    }
                  }
                }
              }
            }
          }
        }
      }
    }
  }
```

This is only a small piece of the API that we are going to be building in the next few sections of this chapter.

The following screenshot shows the documentation that is generated from this specification file:

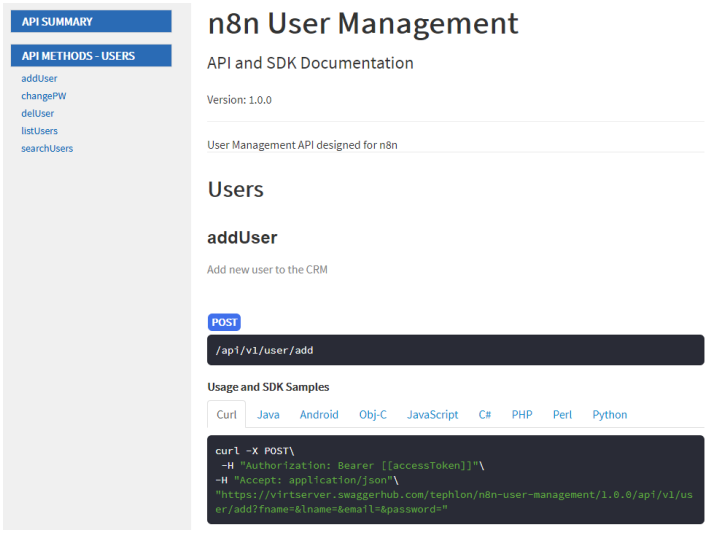

Figure 5.1 – Generated documentation from the OpenAPI Specification file

We have created a full API definition for this, which you can find here:

`https://github.com/PacktPublishing/Rapid-Product-Development-with-n8n/tree/main/Chapter%205`

Now that we know how to design APIs, it is time to dive into n8n and start the basic setup for creating our API.

Configuring the Webhook node to handle requests

The core node for building an API in n8n is the Webhook node. While this may seem a bit strange on the surface, it makes a lot of sense when you start to think of it in the correct frame of mind.

Webhooks are web services that are sitting on a system, waiting to be called upon to perform some action. Meanwhile, an API is a service that a client uses to perform actions on a remote server:

Figure 5.2 – The Webhook node

So, what is seen as an API from the client's perspective is the same as a Webhook from the server's perspective! And this is why we use a Webhook to create an API.

The Webhook node is a trigger node that executes a workflow when it receives a remote connection. It collects the information that it receives and performs actions based on that information.

Parameters

The Webhook node is configured by setting the parameters in the node itself. Each of these parameters modifies the behavior of the Webhook and some parameters, such as the **Authentication** and **HTTP Method** parameters, enable even more options for the user.

Webhook URLs

If you click on the text in the Webhook node that reads **Webhook URLs**, it will open up a small panel below it. You will see two buttons, one reading **Production** and the other reading **Test**. These are the two different types of Webhook URLs that are available to use.

These URLs in n8n cloud are built by combining the n8n protocol (`https://`), the hostname (`tephlon.app.n8n.cloud/`), the Webhook root (`webhook` for production and `webhook-test` for test), and the Webhook path (`f929fdc9-b62a-4661-913e-b5648c407edd`). This creates the two Webhook paths of `https://tephlon.app.n8n.cloud/webhook/f929fdc9-b62a-4661-913e-b5648c407edd` for **Production** and `https://tephlon.app.n8n.cloud/webhook-test/f929fdc9-b62a-4661-913e-b5648c407edd` for **Test**:

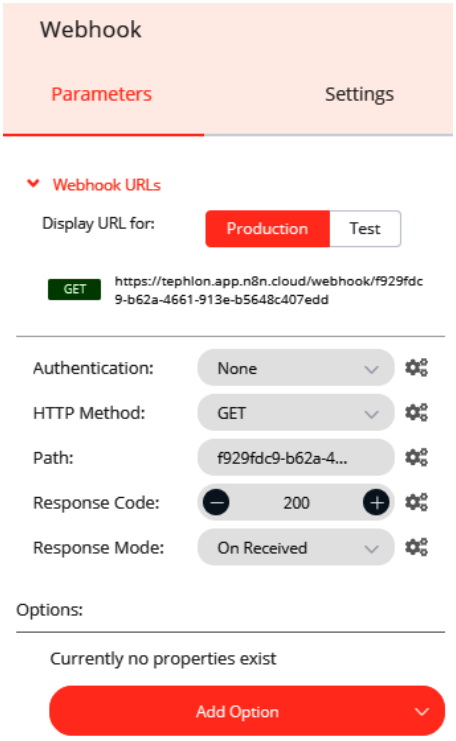

Figure 5.3 – Initial Webhook properties

You can modify the Webhook's endpoint locations by changing the **Path** parameter. If we were to change the path from `f929fdc9-b62a-4661-913e-b5648c407edd` to `api/v1/user/add`, then the Webhook paths would change to `https://tephlon.app.n8n.cloud/webhook/api/v1/user` for **Production** and `https://tephlon.app.n8n.cloud/webhook-test/api/v1/user/add` for **Test**:

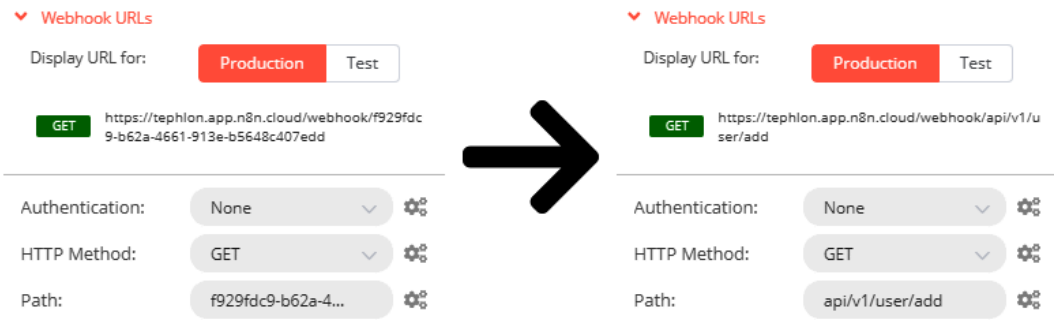

Figure 5.4 – Changing the Webhook paths

The two different types of Webhook URLs – **Production** and **Test** – serve two different purposes. The **Production** URL is used when your workflow has been saved and set to **Active**. The Webhook will then respond even when the Editor UI is not open. It is designed to be available and work completely on its own.

The **Test** URL is designed to be used when you are building and troubleshooting your API. It still requires that your workflow be saved to register the Webhook URL but it will only be active when one of the following conditions has been met:

- You press the **Execute Workflow** button in the Editor UI.
- You press the **Execute Node** button in an open node that is either the Webhook node or a child node to the Webhook node with no cached information available to be processed.

The **Test** URL is available until one of the following actions occurs:

- A connection attempt is made.
- The **Stop** button is pressed in the Editor UI.
- 120 seconds have passed since the Webhook was initiated.

Once this happens, the Webhook URL will unregister and the workflow will stop.

The purpose of the **Test** URL is to provide an easy way to see what is happening in each node when a request is made and troubleshoot the workflow for development.

Authentication

Since APIs are a very common way for people to programmatically access pure data from a system, there is a good chance that you may want to secure that data. Even if your API is going to be publicly available to everyone, it can be a good idea to set up registration and authentication for your API so that you can track who is using (or abusing) your API.

While there are many different aspects of securing your API (many of which will be covered a bit later in this chapter), we want to take a quick look at authentication for your API. This can be accomplished using the **Authentication** parameter.

The Authentication parameter has three different options:

- None
- Basic Auth
- Header Auth

Selecting **None** for the **Authentication** parameter is self-explanatory: the Webhook will not look for any form of authentication before executing the workflow or returning information. While there are valid use cases for not using any authentication for your API (for example, you want to use the Webhook like a web server and display web content), it is generally frowned upon as a practice.

Basic Auth is the simplest form of authentication that n8n can use. It essentially sends a Base64 calculated version of a username and a password to n8n and compares it to the password information that it has on record.

For example, if we were to use Basic Auth for our API and the username and password required are `jim.nasium` and `123456`, the API client would calculate the Base64 version of `jim.nasium:123456` (which is `amltLm5hc2l1bToxMjM0NTY=`) and send it in the header request.

Calculating the Base64 Value

If you ever find yourself in need of generating the Base64 value for a `Function` node, you can use the following code to do this for you. Simply replace the values of the username and password:

```
var username = "jim.nasium";
var password = "123456";
var encoded = Buffer.from(username + ":" +
password).toString('base64');
```

While this is better than no authentication at all, it isn't all that great. Since Base64-encoded text is really easy to reverse engineer, it is recommended that the API is secured properly by complimentary means such as SSL/TLS certificates.

Header Auth is similar to **Basic Auth** in that it sends a value in the header of the API request. However, the difference is that it is just a random string of characters that is very difficult to memorize.

In both the **Basic** and **Header Auth** scenarios, you are required to create credentials that hold the information that's required for users to access the API. Without these credentials, you users will not be able to use the API.

HTTP methods

As we mentioned earlier in this chapter, the n8n Webhook node supports three different HTTP methods:

- `GET`
- `HEAD`
- `POST`

Depending on the HTTP method that you select, different options become available to you.

General options

Four options are available for all the methods:

- Response Content-Type
- Response Headers
- Property Name
- Raw Body

Response Content-Type indicates the media type that will be returned to the user. This is typically a two-part designation, with the first part representing the type and the second part representing the subtype, separated by a slash ("`/`") character.

For example, if your API is returning JSON data, you would set **Response Content-Type** to `text/json`.

For an official list of all the media types that are available, visit the **Internet Assigned Numbers Authority** (**IANA**) *Media Types* web page at `https://www.iana.org/assignments/media-types/media-types.xhtml`.

The **Response Headers** option allows you to add additional key/value pairs to the headers to provide extra metadata to the response. This is information about the data that the user is receiving from the API.

It is often used to confirm that the data is accurate and has not been tampered with between the sender and the receiver. It could contain a hash algorithm that the receiver could calculate to determine that the information is accurate or a timestamp to indicate when the information was received.

The **Property Name** option allows you to return just the value of a specific value/key pair.

For example, let's assume that you have a Webhook set up that outputs the following JSON object:

```
{
  "response": "Hello!",
  "status": "Successful"
}
```

Now, if we were to modify that Webhook and add the **Property Name** option with a value of **status**, our output would be as follows:

```
Successful
```

This is useful if you have HTML in a key-value pair and you just want to output the HTML to display a web page.

The final option is **Raw Body**. The **Raw Body** option is a binary (off or on) value that indicates the information coming into the Webhook is in a raw format, such as XML or JSON.

Additional POST option

When you select **Post** for the HTTP method, there is one extra option available. This is the **Binary Data** option. This option is used to indicate that the API is expecting that there will be binary data attached to the request. This is handy when you are uploading files to the system.

Response Code

The **Response Code** parameter is used to reply to the sender with a quick response, letting the requester know the results of their request.

We went through response codes in detail earlier in this chapter, so I'm not going to go over them again, but make sure that you reference the code and reply with the appropriate response.

> **Note**
> The response code that you are selecting is going to be sent when everything goes well. Other failure response codes (for example, `404`) will be sent by the system and do not need to be programmed here.

Response Mode

The **Response Mode** option controls how the Webhook responds to the request. Two options can be set for the Response Mode option:

- On Received
- Last Node

In general, if you are not passing data back to the requester, you will want to use the **On Received** option. Otherwise, if you are sending information back via the API, you should use the **Last Node** option.

The **On Received** option will immediately send the value of the **Response Code** option back to the sender. It does not wait for the workflow to complete before returning this code.

Things get a little bit more interesting with the **Last Node** option. When you want to return information to the requester, this option will initiate a workflow and then the results (or part of the results) of the last executed node will be returned to the requester as JSON.

Response Data options

When you select the **Last Node** option for **Response Mode**, this enables a new parameter called **Response Data**. The three options for **Response Data** are as follows:

- All Entries
- First Entry JSON
- First Entry Binary

To understand how this option works, we will look at an example. Let's assume that we have a workflow with the final node outputting the following JSON array:

```
[
  {
   "name": "Jim Nasium",
    "city": "Berlin"
  },
  {
    "name": "Kris P. Bayken",
    "city": "Edmonton"
  }
]
```

Also, for each JSON object in the array, there is a binary file attached to the object; `CV - Jim Nasium.pdf` for the first one and `Resume - Kris P. Bayken.pdf` for the second one.

If we set the **Response Data** parameter to **All Entries**, the requester will receive the entire array contents but none of the files.

When we change **Response Data** to **First Entry JSON**, the receiver will receive the following JSON object:

```
{
  "name": "Jim Nasium",
  "city": "Berlin"
}
```

Finally, if we use the **First Entry Binary** option, the requester will be sent the `CV - Jim Nasium.pdf` file.

Now that we have gone through the Webhook node in great detail, it is time to start building out our API in n8n.

Building the API in n8n

We are finally ready to start building in n8n! Let's start by outlining our project.

API project specifications

For our project, we are going to build a simple user management API that will modify the user information in an Airtable database. Most of the workflow that we will be building isn't important to the API itself. All we have to be aware of is the result of each API.

Our API is going to have five endpoints:

- `POST /api/v1/user/add`
- `GET /api/v1/user/list`
- `GET /api/v1/user/search`
- `POST /api/v1/user/delete`
- `POST /api/v1/user/changepw`

The base URL for our API will be `https://tephlon.app.n8n.cloud/webhook` since that is the Webhook URL for my n8n cloud instance.

Our API will also use header authentication with a bearer token value of `675tryfhgui89765tyrfghjui89765uyr4thfgjuio`.

The Airtable database will store the following information:

- **First Name**: The user's first name
- **Last Name**: The user's last name
- **Email**: The user's email address
- **Password Hash**: A calculated value representing the user's password

> **Note**
> The reason that we do not store the user's password is for security. If the data table were to become compromised, the user's password would be out in the wild. By storing a calculated value based on the user's password, the same calculation can be performed each time the password is supplied to n8n, and the calculated value is compared to the stored value.

Creating credentials

The first step will be to create the header authentication:

1. In the n8n Editor UI, click on the "^" icon and click on **New**.
2. When prompted, select **Header Auth** for **Credential type**.
3. In the **Create New Credentials** form, enter the following information:
   - **Credentials Name**: `Header Authentication`
   - **Name**: `Auth`
   - **Value**: `Bearer 675tryfhgui89765tyrfghjui89765uyr4thfgjuio`
   - **Nodes with Access**: `Webhook`

   The final credentials form should look like this:

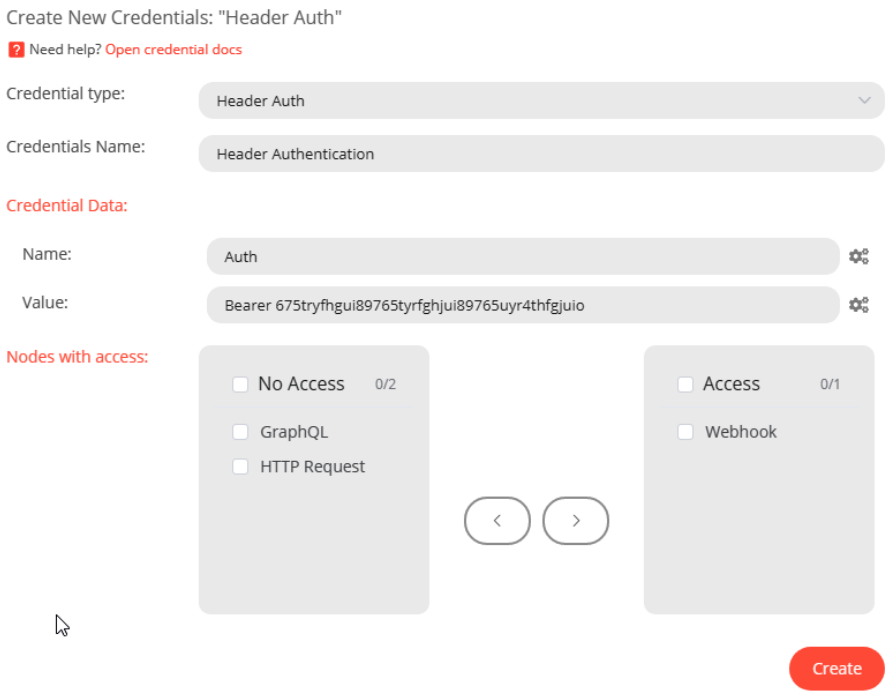

Figure 5.5 – Header Auth credentials

4. Click the **Create** button to complete this process.

Creating Webhooks

Next, let's create the Webhooks in n8n:

1. Create five new Webhook nodes and name them like so:
   - `Add`
   - `List`
   - `Search`
   - `Delete`
   - `ChangePW`
2. Configure each of the Webhooks based on the following table:

Name	Header Auth	Authentication	HTTP Method	Path	Response Code	Response Mode	Response Data
Add	Header Authentication	Header Auth	POST	api/v1/user/add	200	Last Node	First Entry JSON
List	Header Authentication	Header Auth	GET	api/v1/user/list	200	Last Node	All Entries
Search	Header Authentication	Header Auth	GET	api/v1/user/search	200	Last Node	All Entries
Delete	Header Authentication	Header Auth	POST	api/v1/user/delete	200	Last Node	First Entry JSON
ChangePW	Header Authentication	Header Auth	POST	api/v1/user/changepw	200	Last Node	First Entry JSON

That should complete the initial Webhook creation. Make sure to save and activate the workflow.

The rest of the workflow

While the rest of the workflow is important for generating the proper outcomes for the API, it has very little to do with the API itself.

You can download the whole workflow here: `https://github.com/PacktPublishing/Rapid-Product-Development-with-n8n/blob/main/Chapter%205/User_Management_API.json`.

Now that we have built the API in n8n, let's look at how to add further security to your API endpoints.

Securing your API endpoints

Earlier in this chapter, we talked about using Webhook authorization to help secure your API. This is an important first step but that does not mean that your API is secure. There are several extra actions that you can take to secure your API.

Using SSL/TLS security

SSL and its younger brother, TLS, are common security standards on the internet. Any time you see `https` in front of a website address, this means that one of these systems is at work.

The purpose of SSL/TLS is to encrypt the communication between the client software (for example, the web browser, n8n, and API client) and the web server that is hosting the API. This way, if someone were to set themselves up between you and the API server, all they would get is jibberish instead of the API tokens and passwords that are used to access that information.

Limiting where users come from

One way of reducing the probability of being taken down by a malicious actor is to limit who can access your API based on where they are coming from. For example, if all your clients who use your API are from Germany, then there is no point in allowing people from Canada to access the API.

While this is not a perfect solution, since it is relatively easy to get around by using a VPN, this means that people who are out to cause trouble need to jump through yet another hoop.

If you want to take this option to the next level, you can even limit access to a single IP address so that only people from a specific office can use your API.

Proxying your API

Putting your API behind a proxy is one of the smartest things that you can do. Here are some of the benefits of doing this:

- It prevents people from directly accessing your API server.
- It hides other potentially dangerous open ports.

- It can reduce calls to your API and other tools by caching requests and returning results without having to talk to the API server.
- It can distribute requests across several servers.

With all of these advantages from the proxy, it is well worth looking into setting up a proxy.

Rotating security tokens

With security breaches happening daily, it is well worth changing the API tokens regularly as well. There are several different ways to do this but you need to find a way that works for you.

One of the best ways to do this is to have a token expire after a certain period. At that point, the user would be required to get a new token. This can be automated so that as soon as the token times out, the system knows to reach out and request a new one.

This way, even if your token were to get out and someone on the internet were to start using it, they would only have access for a short period before the token would expire and then require a new token to be generated.

Tracking and limiting the number of requests

It is important to record every single connection attempt that occurs on your API server. This provides you with two advantages:

- You can identify if there is any strange behavior occurring on your API server.
- You can prevent one person or organization from abusing the API server.

Users (as identified by their token) and organizations (as identified by their IP address) should have the number of calls they make throttled to a reasonable speed.

Providing metadata in your API responses

Embedding useful metadata in the information that you provide to your users is a great way to allow the end user to verify the information that they are receiving is accurate. Some examples of good metadata are as follows:

- Providing an **MD5** sum value when downloading a file for the user
- Indicating the record count being returned from a query
- Performing a GeoIP lookup of the city the user is coming from so that they can confirm the request is legitimate

- Returning the length of the results in a certain number of characters

Once your API is secure, it is time to start testing its robustness.

Testing your API

Creating a testing plan for your API is an important part of the design and rollout process. If you do not test your API properly, there is no telling what results your clients will get.

Because of its importance, let's look at some recommendations for API testing.

Use a testing platform

While you may be able to manually test some small or simple APIs, the bigger they become, the harder they are to test. Not only does this allow you to do more for testing, but it also allows you to easily retest after you make changes, knowing that the test is the same as it was the previous time.

If you don't have a testing platform, you can easily create several testing scripts using the cURL command-line tool or use something such as the documentation and testing tool built into Insomnia (`https://insomnia.rest`).

Follow the documentation

The documentation that you have created or been provided with is very important. Make sure you can do everything that the documentation says that it can do and none of the things that it doesn't say you can do.

You should be able to do absolutely everything that is in the documentation without fail.

Try to break it

This is the part of the testing process that you can have a lot of fun with. Here are some ideas I've had success with in the past:

- Send information in different character sets.
- Enter SQL commands as usernames or passwords and see if the commands execute on the server.
- Overload the server by making thousands of calls per second.
- Try to get access to the information you shouldn't be able to on the server.

Confirm the data

When you are testing the API, make sure that the data you are expecting to receive is the data you are receiving. It is really easy to look at one or two samples and then extrapolate over the entire dataset that the information being returned is accurate.

Ongoing testing

Just like data backups should be tested regularly to ensure that the data is recoverable, it is also important to continue testing your API regularly.

As the environment that the API is running in changes, so can the performance of your API. Some environmental changes that could cause your API to begin behaving poorly include the following:

- A sudden increase in the number of users
- Increased database size
- Larger and more frequent data requests
- A poorly designed proxy
- Changes to the work schedules

By continually testing and upgrading the API, you can continue to ensure that the API performs as expected and continues to keep your clients happy.

Summary

In this chapter, we showed you how to plan out your API and build it in n8n using Webhooks. We also covered how to test and secure your API so that it functions properly when it is in production.

While it may feel as though we have covered a lot about APIs, there are still considerable amounts for us to go through.

One of the aspects of APIs that I did not go into very much detail about was manipulating the data that's sent and received by APIs in the workflows. If APIs are a language that computer systems use to communicate with each other, then the data manipulation that occurs inside an n8n workflow is the translation of that API language so that two or more different systems can communicate, even when they don't know each other's language.

We are going to be talking about this data manipulation in more depth in the next chapter as it is not just an important topic – it is a critical one.

6
Powering Your API with a No Code Database

In this chapter, you will learn to work with no code databases for data storage. You will learn about no code databases, selecting a database for your project, and reading and writing to **Airtable**. You will also learn about some of the best practices when working with these databases. The concepts learned in this chapter will help you to use a data store for your projects to store user-generated data and build a complete product.

This chapter will cover the following main topics:

- Learning about no code databases
- Selecting a database for your project
- Using **Airtable** for reading and writing data
- Best practices for working with databases
- Optimizing your **application programming interface** (**API**) for production

Technical requirements

Here is a list of technical requirements that you'll need to prepare before continuing with the chapter:

- You have created an account on **Airtable**
- n8n is running and the Editor **user interface** (**UI**) is open

Learning about no code databases

Databases often form the backbone of products. Databases are generally systems that store large amounts of data. The user can add, delete, or modify data while also viewing and performing calculations on that data.

Databases come in many different forms, such as **Structured Query Language** (**SQL**) databases, NoSQL databases, and time series databases. Depending on the use cases, it often makes sense to choose one over the other. A lot of these databases use query languages to be able to interact with the database to conduct basic queries such as inserting, reading, updating, and deleting data. You can see some database examples in the following screenshot:

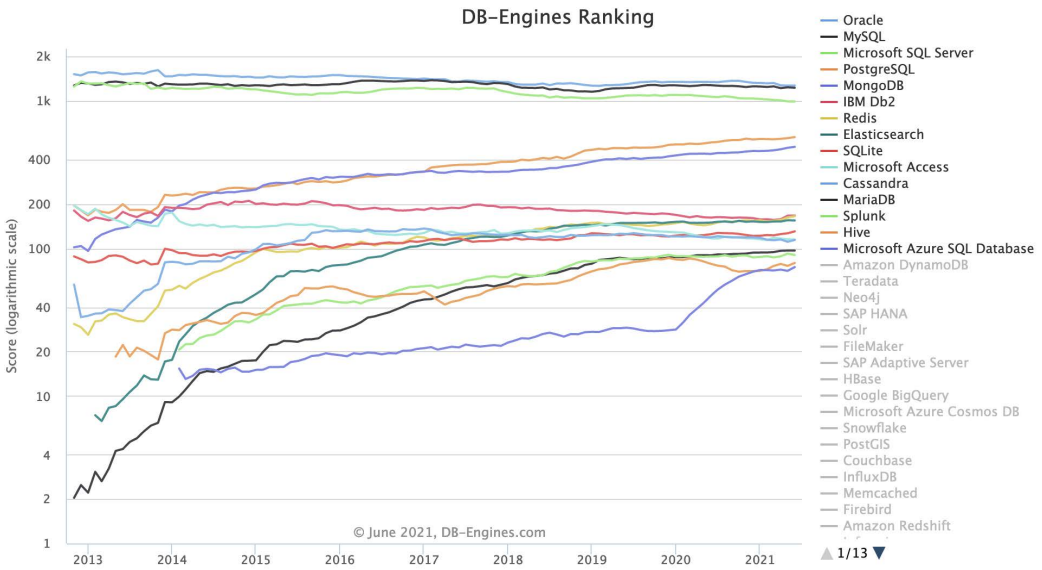

Figure 6.1 – Historical trend chart from db-engines.com

Since we are focusing on building a product with no code tools and we don't have the time to learn these query languages quickly, we'll have to look for some alternatives. Luckily, there are already a number of great alternatives that we can choose from. Two of the most popular ones are **Airtable** and **Google Sheets**. These tools are easier to use as compared to traditional databases as they employ a familiar spreadsheet-like design that is easier to comprehend than database models. Since these tools are rather robust as well, more and more people have started using them in their projects.

Both **Airtable** and **Google Sheets** have an n8n node, and we can use these nodes to perform **create, read, update, and delete** (**CRUD**) actions to power our API and application. Now that we have an idea about what no code databases are, let's think about how we can choose one that we can use during the duration of this book.

Selecting a database for your project

Apart from **Google Sheets** and **Airtable**, there are a number of really cool no code databases such as **Baserow**, **Supabase**, **SeaTable**, and **NocoDB**. How do we select the right database for our product? There are a couple of questions that you can ask yourself that might help in making this decision easier:

- What do I want the database to do for my project?
- Does this tool have an n8n node or at least an API?
- How active is the community and support ecosystem for this tool?
- What's the level of maturity of the product? Is it stable? Has it been in the market long enough to be battle-tested by users?
- Are educational resources about the tool widely available? What's the learning curve like?

While choosing the no code database to include in this book, we asked ourselves the same questions. We decided to go with **Airtable** because of the following reasons:

- **Airtable** has a large community and an incredible support system.
- **Airtable** has a decent API with an easy authentication system. n8n has a well-documented **Airtable** node too!
- **Airtable** has been around for a bit, and a lot of makers use it to build their projects.
- **Airtable** puts out a lot of educational content regularly and it's straightforward to get started with.

The following screenshot provides an overview of the **Airtable** database:

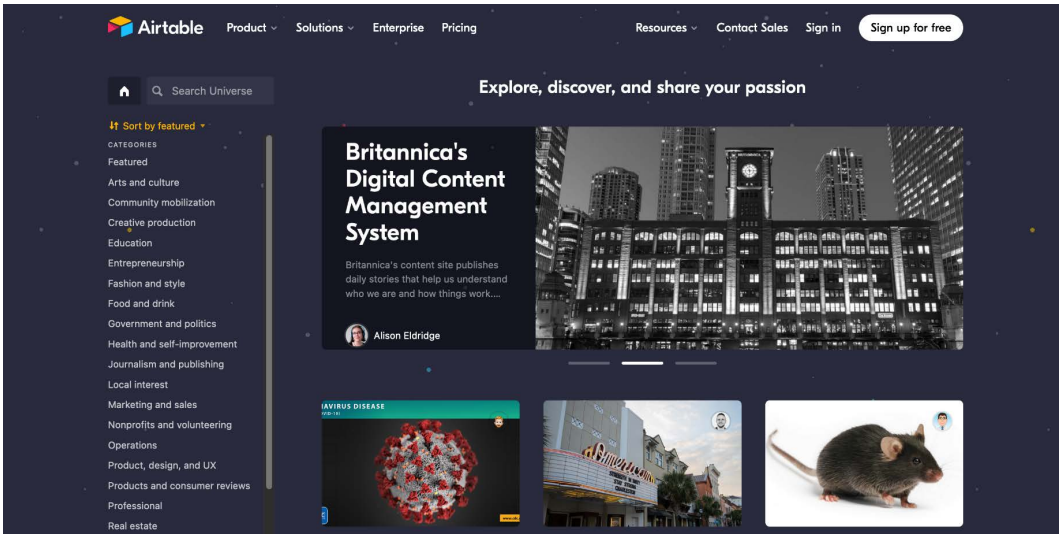

Figure 6.2 – Airtable has a lot of educational resources and a strong community

While you are asking yourselves these questions, it is quite possible that your answers might be different depending on the project that you are planning to work on. That's okay! Each tool has its own niche and superpowers that might make it more suitable for a certain type of project.

Now that we've made sure that **Airtable** is the correct choice for us right now, let's take a look at how we can use it from n8n workflows.

Using Airtable for reading and writing data

Let's start off by creating a new base from scratch in **Airtable**. I am going to name it `The n8n book`. Edit all the existing fields so that we have the following four fields with the single-line text field type:

- **UserID**
- **First Name**
- **Last Name**
- **Email**

This is very similar to the table into which data was being inserted in the previous chapter. For the sake of brevity, we have taken out the **Password Hash** field. Your table should now look like this:

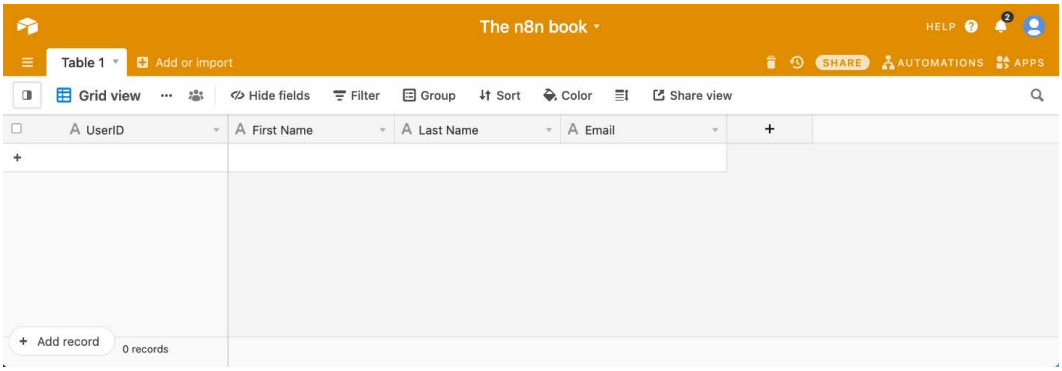

Figure 6.3 – This is what your Airtable table should look like

Now that we have prepared our table, let's pop over to n8n's Editor UI and follow these next steps:

Create a new workflow and add a **Set** node to it. We'll need the **Set** node to make sure that we send well-formatted data to the **Airtable** node.

Open the **Set** node and add four values of the `String type`. For each of these four values, do the following. In the **Name** field, enter the same as the name of the columns in the **Airtable** table. Toggle the **Keep Only Set** button to `true` (green). In the **Value** field, enter anything you like.

Click on the **Execute Node** button, and your **Set** node should look like this:

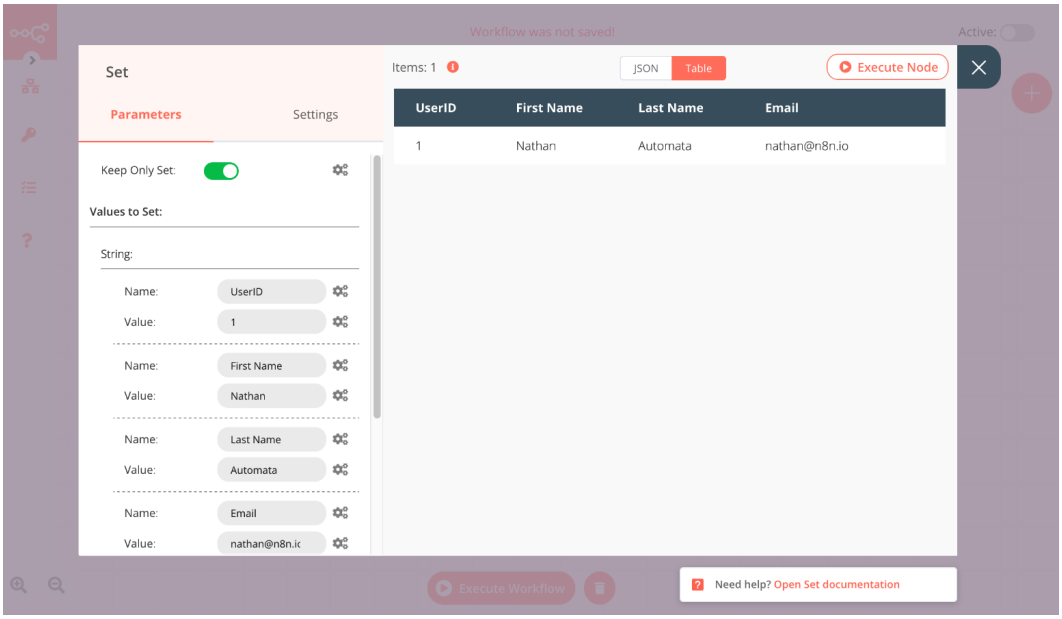

Figure 6.4 – Output from the Set node after following the aforementioned steps

Now that we have structured the data in a way that would correspond to the columns in **Airtable**, let's add the **Airtable** node and connect it with the **Set** node. Here's what your Editor UI should look like at this point:

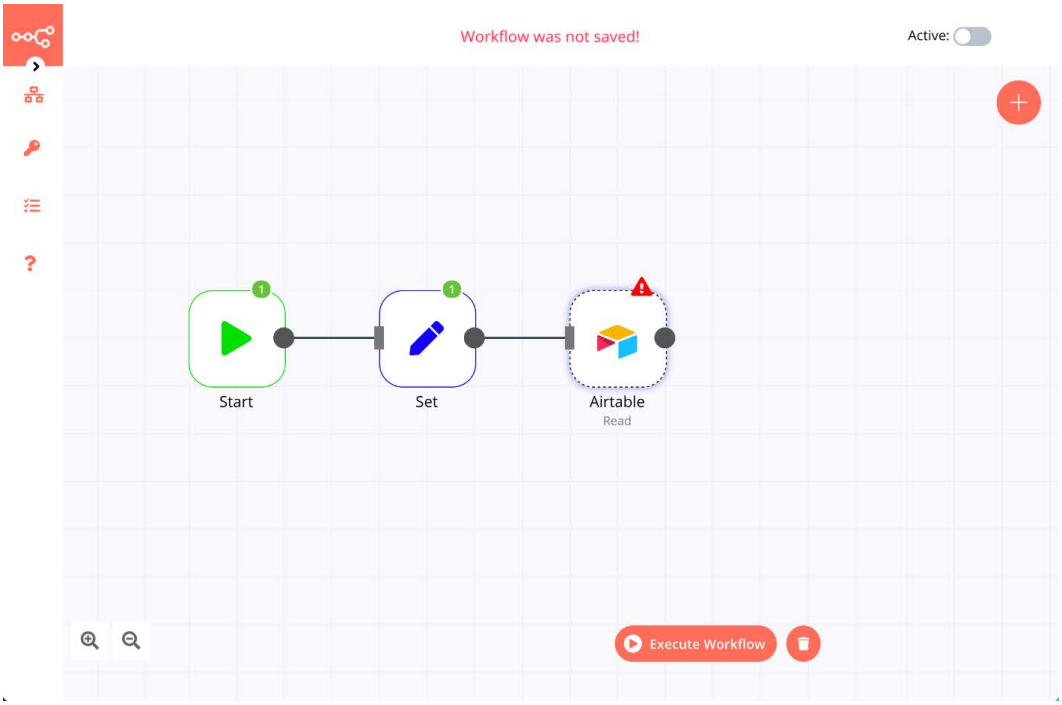

Figure 6.5 – The workflow should look like this at this point

Open the **Airtable** node and enter your credentials. You can find the API key by following the steps mentioned on this page: `https://docs.n8n.io/credentials/airtable/`.

Change the **Operation** field to `Append`, since we want to insert the data from the **Set** node to our **Airtable** table. We now need to acquire the **Base ID** value. Head over to the API page in **Airtable** (`https://airtable.com/api`) and select the base that you created. You'll find the **Base ID** value there. Paste it in the **Base ID** field in n8n. Enter `Table 1` in the **Table** field and click on the **Execute Node** button. It should now look like this:

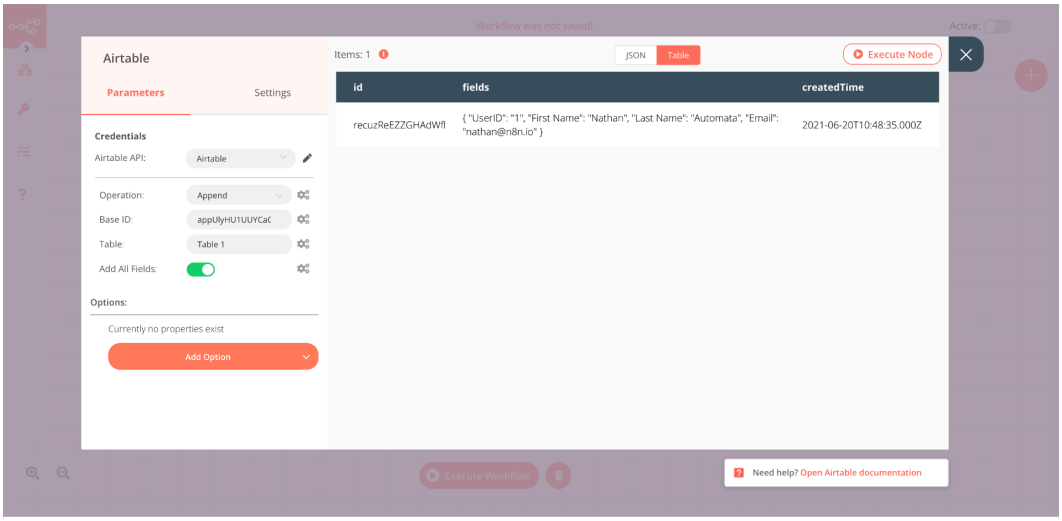

Figure 6.6 – Output from the Airtable node after following the aforementioned steps

If you go back to your **Airtable** base, you'll notice that the record has been added to the table by this n8n workflow. Congratulations—you've just added your first record to **Airtable** using n8n! Here's a screenshot that showcases how your **Airtable** table should look at this point and how it relates to the data that we structured in the **Set** node:

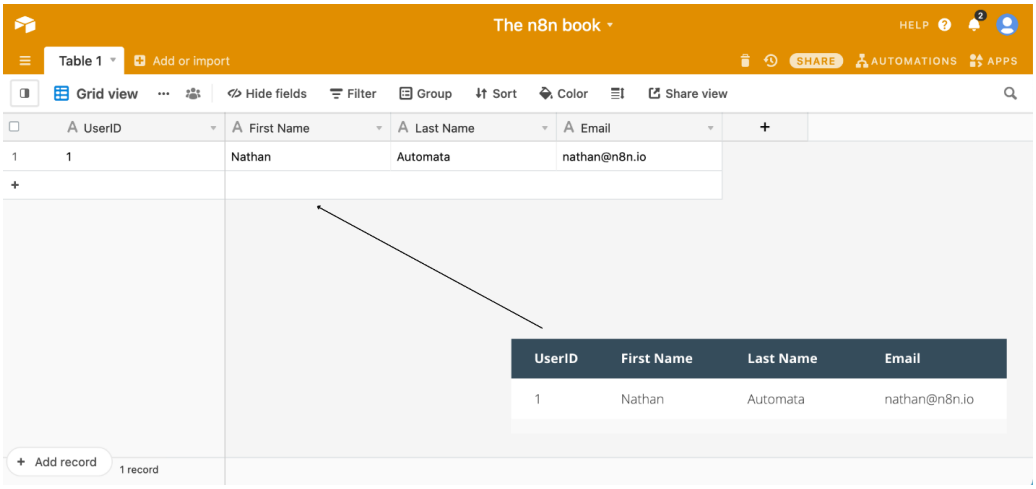

Figure 6.7 – Record inserted into the Airtable table and how it related to the data from the Set node

You can replace the **Start** node with other nodes to source the data that you want to add to **Airtable** and make the appropriate changes to the **Set** node. You might remember from the last chapter that we got the data from the API that we created using the **Webhook** node.

This combination of the **Set** node (to structure the data in a form expected by the database) and the **Airtable** node (to insert the data into **Airtable**) will remain the same across workflows. In case you want to use another database for some other project, you'd replace the **Airtable** node with that node.

Here are some key things to keep in mind about inserting data into databases:

- Spreadsheets and databases have columns, such as **First Name** and **Last Name**. The data you send to a database node needs to match these column names for each row of data that you want to insert.
- A lot of times, you might not get data from APIs in a form that works best for you. In those cases, the **Set** node can help you remodel the data that you need according to your database's columns and discard the data that you don't need.
- Spreadsheet and database nodes in n8n perform their configured action (such as `Append`) on each item of input data.

Reading data from an **Airtable** table is relatively straightforward. To do that, create a new workflow and connect the **Airtable** node to the **Start** node. Select `List` as the operation and enter the same credentials and **Base ID** value as the previous workflow.

Click on the **Execute Node** button, and the output should look like this:

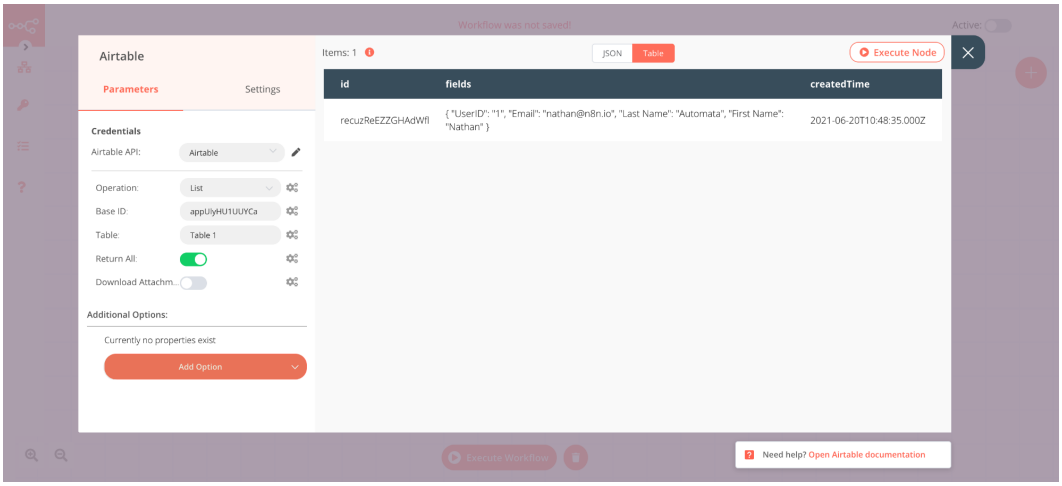

Figure 6.8 – Output of the List operation of the Airtable node

Now that we have listing data from **Airtable** out of the way, let's learn how to update a record that already exists in **Airtable**. You might have noticed that the `List` operation of the **Airtable** node returns an **identifier** (**ID**) as well. Each record in **Airtable** has a unique ID as well as a timestamp of when it was created. The ID is especially useful for operations such as `Update` and `Delete`.

Consider this scenario: You need to find a record where the first name is `Nathan` and update the last name of the person to `Automaton`.

You can then build a workflow like the one shown in the following screenshot to update the particular record in the **Airtable** table:

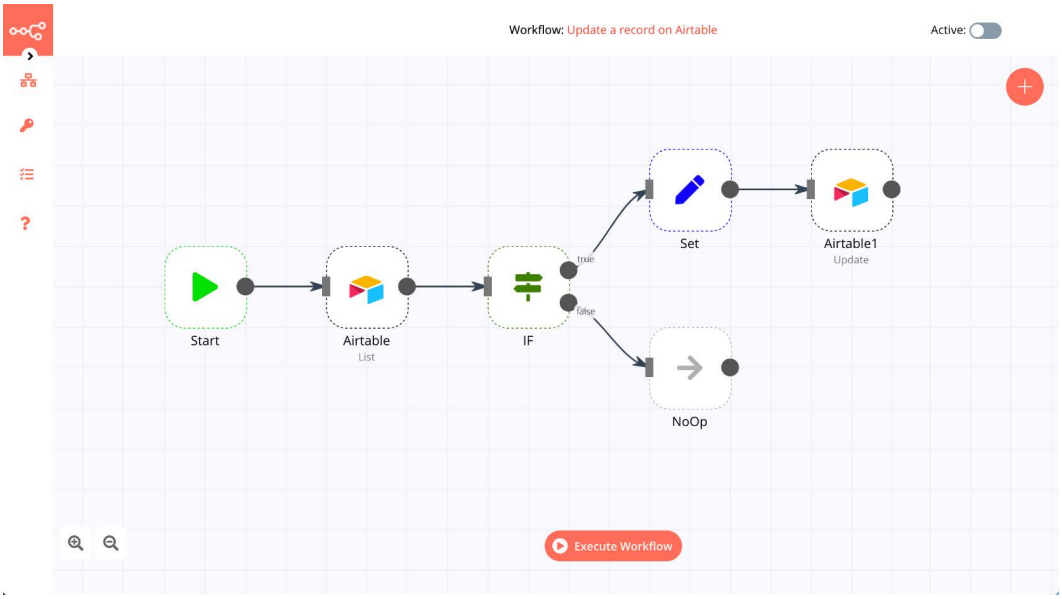

Figure 6.9 – Updating a particular record in Airtable using an n8n workflow

The first **Airtable** node lists all the records that are present in the table. The **IF** node checks whether the first name of the record is `Nathan`. If it isn't, n8n goes to the **NoOp** node, and nothing happens. If the first name is found to be `Nathan`, we use a **Set** node to add the new value for the **Last Name** field. Here's what the **Set** node looks like after configuration and execution:

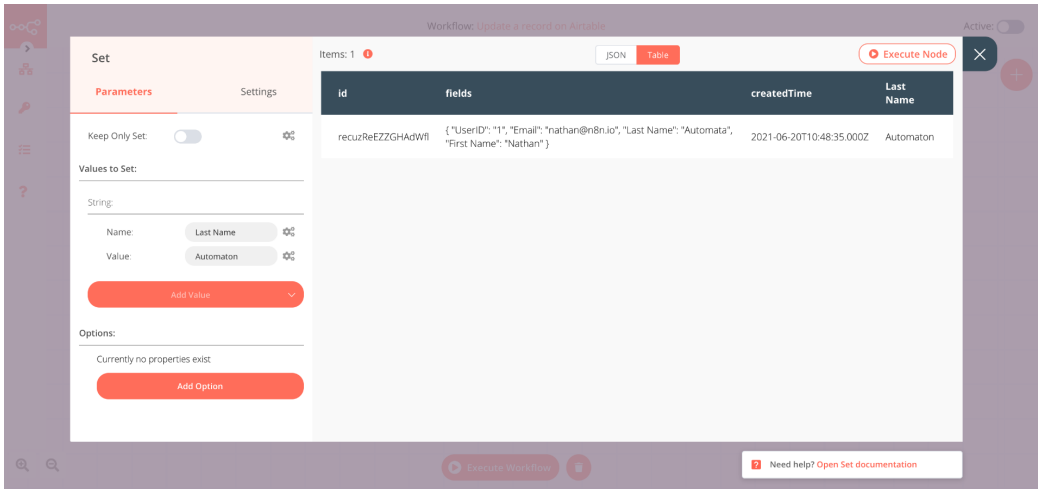

Figure 6.10 – Configuring the Last Name field for updating the record with Nathan as the first name

And finally, we have the **Airtable1** node, which will update the record. We have used the ID of the record to be updated from the first **Set** node (we originally got it from the **Airtable** node) as well as specifying that only the **Last Name** field should be updated. You can see the **Airtable1** node here:

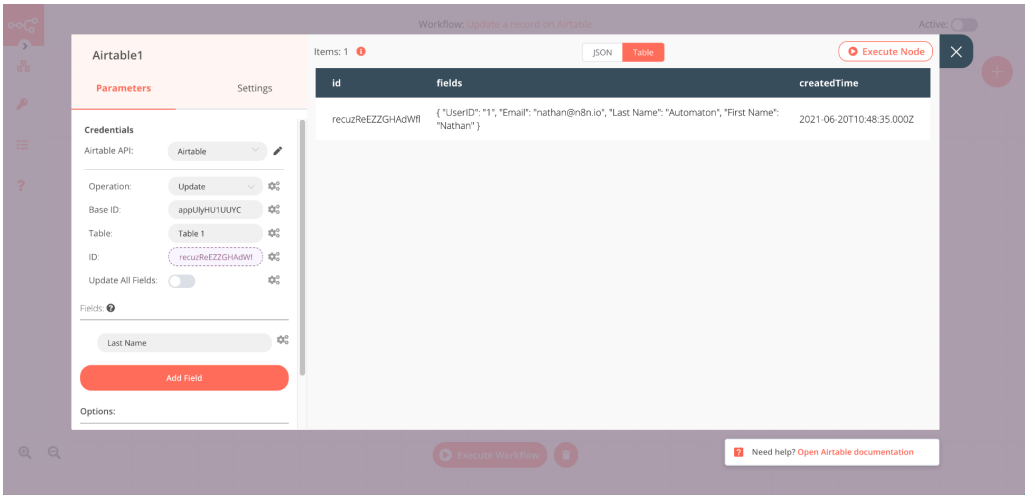

Figure 6.11 – Configuring the Airtable1 node for updating the Last Name field of the record

And voilà! The workflow has updated the **Last Name** column for the specified record in **Airtable**. You can use a similar workflow to delete records as well.

When using this workflow as part of an API, you can get values such as the text for which to perform a lookup and which column to look in, as well as the updated record using the **Webhook** node. You can then use expressions to make sure that the API endpoint with this workflow can handle dynamic requests without having to create specific workflows for different columns. Now that we know how to work with no code databases using n8n, let's learn about some best practices for working with databases.

Best practices for working with databases

There are a lot of different aspects that go into working with databases that you will generally use in an enterprise environment, but things can be a bit different with no code databases. Because of how these databases are designed, built, and hosted, we need to think a bit differently when we use them.

Let's take a look at some of the best practices around working with databases and how you can use them in a way that is both effective and efficient.

Minimizing bandwidth

While it is not always the case, no code databases are generally hosted somewhere on the internet. This means that you do not have as much bandwidth available between you and the database that you would use if the database were hosted on the same network, which is the case for traditional databases.

Because this bandwidth is now at a premium (and, depending upon how your database is hosted/priced, you may literally be paying for every byte that you send and/or receive from the database), it is very important that you make sure you use it wisely.

Compressing data

Often, the data that is stored in databases is text-based. Text data has a very high compression ratio, which reduces how much information needs to be sent or received between you and your database.

If you have the ability to compress data between the two systems, this will increase the speed of your transactions and reduce your bandwidth.

However, keep in mind that this will also increase the **central processing unit** (**CPU**) utilization on both n8n and your database as the compression needs to be calculated on both sides.

Minimizing API calls

API calls to your database, such as bandwidth, can be expensive (literally, if you are paying per API request). They take up resources, slow down your application (as the application needs to wait for the call to complete or time out), and increase dependencies on the database.

If you do everything you can to reduce API calls, these issues can be minimized or, in some cases, avoided altogether.

Minimizing database queries

Because databases generally work very quickly, it is easy for us as developers to become lazy when accessing a database. Why bother modifying the code to write three records to the database with a single query when it is easier to write each record on its own in three separate queries?

This type of programming may work with large databases that are sitting on the same network as you, but when they can be located on the other side of the planet over a fluctuating internet connection, these queries must be optimized and minimized.

Minimizing database writes

While it is one thing to read data from a database, it is a completely different thing to write data to a database. Write operations tend to consume significantly more resources than read operations on a database. Plus, they take longer to execute, often because data needs to move around in memory, or even sometimes on disk.

Because the cost of writing to your database is relatively expensive, only write data when you absolutely have to and write as much data as possible each time you do write. This will give you the most bang for your buck for each database write.

Enabling data caching

If we are truly serious about minimizing the amount of data we read and write between n8n and the database, a great strategy is to store a copy of select database tables locally in n8n, either in memory or in a local **JavaScript Object Notation** (**JSON**) file.

Then, each time a request needs to be made to the database, you can first ask whether the database has changed since the last time the cache was updated. If it has, then n8n should pull down just the changes that were made to the database and write those to the cached data. If there were no changes, then updating the cache can be skipped and the query can be executed locally.

This can be a significantly more efficient way of looking at the data in a database than querying the database each time. It not only speeds up your database lookups but also reduces the number of calls to the database in the long run as well.

Backing up the database

Your database is the core of your application. The database must be available as much as possible and, if the database is lost, you have a way to recover it.

Your best line of defense is to back up your database as often as possible. This way, you will minimize the amount of data loss and reduce the amount of time it takes to recover.

Recording transactions

Another way to ensure that you can recover from not only data loss but also data overwrite is to record every transaction that occurs on the database to a separate transaction table. This way, if there is a gap between the time your database became unavailable and the time of the last backup, you can recreate those transactions.

Also, if your database backup is completely lost, you can still recover from the data loss by executing the transactions again. This is significantly slower than recovering from backup but infinitely better than losing all of the data.

Using record references and table views

When you are using data that needs to be entered several times, it is more efficient to create a separate record in a different database table with a unique record ID rather than repeatedly writing the same data each time.

For example, if I was creating a database that needed to refer to user information such as first name, last name, and address, I could just write all this information to the database table each time it was needed. But if we were to write this information to a user table, we could then just reference the **UserID** value stored in the table and write that rather than the entire record.

This allows you to reduce the amount of information that gets transferred and minimizes the size of the write.

Securing your database

Because these databases are hosted on the internet and can generally be accessed from anywhere on the internet, it is extremely important that they be properly secured. Ensure that all credentials and API keys are stored securely and are not hardcoded into your applications.

Also, make sure all transactions are over an encrypted (**HyperText Transfer Protocol Secure**, or **HTTPS** for short) connection and, if possible, only allow specific **Internet Protocol** (**IP**) addresses to talk with the database.

Performing calculations on the database

Your database will most likely be a lot more powerful than the system you are using to host n8n. Because of that, if it is possible, get the database to perform calculations, especially if the goal is to provide summary statistics of the data that is already on the database.

Rather than sending all of the data to n8n for processing, perform the processing on the database using query functions such as `COUNT`, `MIN`, and `MAX`. This moves the calculations (that is, CPU load) over to the database and reduces the amount of information that needs to be transmitted between the database and your application.

Load testing the database

It is not uncommon for developers to build an application and it runs just fine in both development and testing, but once it goes into production, the database cannot handle the load due to resource constraints (for example, CPU maxed out; storage too slow; bandwidth constraints).

Make sure that you have a way of load testing the database before it goes into production. This way, you will be able to ensure that the database has all of the necessary resources before it becomes a problem.

Now that our database is ready to go, let's take a look at how we can design and build the API to provide the best performance to users.

Optimizing your API for production

Your API can be one of the most important parts of your application since it is one of the primary ways that your clients read and write data. Because of this importance, it is vital that your API is ready for production right from the start.

Here are some of the ways you can make sure your API is production-ready.

Reducing database calls

Very much in line with the best practices for databases, the fewer times you need to read and write to the database, the better your application will perform. Use many of the strategies mentioned in the previous section to accomplish this.

Caching data before the API

If you have relatively static data behind your API, one of the tricks you can use is to put a caching system in front of your API, which will allow you to give out the information requested by the users without actually touching the API itself. The caching system updates itself with information from the API on a regular basis, and if it determines that there has been no change in the data given out by that API based on the same call being made to the API, the caching system will just send back the data that it has stored locally without bothering the API.

Minimizing API calls

Some APIs require you to have information that is in the database to make another query to the database. A good example of this is user accounts. Generally, you would use the API to query the user table to find the user that you need. Then, you would query the API again, looking for specific information for that user.

What you could do instead is to keep a local copy of the user table in your application and use that to look up user IDs. Then, with that information already in hand, you can query your API only once to get the information you need.

Requiring authentication

While there are a lot of open APIs out there on the internet, it is very important to require authentication, even if you are giving the API service away for free. This increases the level of responsibility that the user has and reduces the likelihood that the API will be abused.

In the event that someone is abusing your API, authentication can make it easier to track down that individual and resolve the issue with their system or stop them from interfering with your API.

Encrypting API data on the wire

Unless your data is encrypted on the wire (that is, when it is being transmitted between the application and the database), API keys, credentials, and sensitive data are susceptible to being eavesdropped upon.

The easiest way to secure this data while in transit is to have a **Secure Sockets Layer** (**SSL**) certificate installed on the server that is providing the API. This encrypts the data and greatly reduces the possibility of someone listening to your API's conversations.

Tracking API requests

It is guaranteed that if you make an API available on the internet, there will be someone out there who is looking to abuse that API. This is why it is critical to keep a log of all API transactions in the event that you need to deal with one of these abusers.

Ensure you track, at a minimum, the following information:

- Timestamp
- API call
- Parameters
- IP address
- Authentication key

This will provide you with the base information to perform some analytics on the data and determine who the abuser is and where they may be working from.

Tying API users to IP addresses

In line with the previous item, if you can get your API to only allow a user to access it from a specific IP or IP range, this can reduce the likelihood that your API will be abused and will allow you to help your users in the event that their information has been compromised.

Limiting the number of API calls per user per second

If you put limits on how quickly users can access the API, this will help to distribute resources more evenly to all your users, along with reducing the likelihood that a user will inadvertently perform a **denial-of-service** (**DoS**) attack on your API.

Properly documenting the API

If you have proper API documentation available for your users and developers, it will be easier for them to use your API properly and keep your error logs clean. This will increase customer satisfaction and reduce the stress on your support team, who need to help these people.

Summary

In this chapter, we learned about no code databases, choosing a no code database for your project, reading and writing to **Airtable**, as well as some best practices when it comes to working with databases and optimizing APIs for production. The concepts learned in this chapter will help you to use a data store for your projects to store user-generated data and build a complete product.

In the next chapter, we examine how you can transform your data inside n8n workflows. We will do some hands-on exercises on sharing data between different n8n workflows, merge datasets from different tables, and also perform some analytics and calculations on these datasets.

7
Transforming Your Data inside a Workflow

In this chapter, you will learn how to manipulate data within workflows so that the APIs that you create can return the data in a useful format. You will also learn about sharing data between workflows, working with arrays and JSON objects, merging datasets, and performing analytics and calculations.

This chapter will cover the following main topics:

- Sharing data between workflows
- Merging datasets
- Performing calculations and analytics

Technical requirements

The following are the technical requirements that you'll need to prepare before continuing with this chapter:

- You should have created an account on Airtable.
- n8n should be running and the Editor UI should be open.

You can find the completed code examples for the chapter on GitHub at `https://github.com/PacktPublishing/Rapid-Product-Development-with-n8n/tree/main/Chapter%207`.

Sharing data between workflows

When you're building workflows with n8n, you may find yourself repeating certain patterns. Examples of such patterns could be pushing data to Airtable, transforming the data to fit a particular format, or performing checks regarding the validity of the incoming data. At other times, your workflows might grow in size with more than 20 or 30 nodes, and it might become difficult to manage so many different nodes and logic in a single workflow.

If you come from a programming background, you can probably relate this to creating functions or modules so that you can create reusable chunks of code that are modular and easier to manage. n8n allows you to do this using the **Execute Workflow** node.

Let's consider a workflow: we need to get data using the **Hacker News** node, filter the data in the workflow to include only the title and the URL of the articles, and insert the data into an Airtable. Let's break this workflow down into two parts for illustration purposes:

- Getting the data from Hacker News
- Filtering the data and inserting it into Airtable

To do this follow these steps:

1. Open the n8n Editor UI and create a new **Workflow**. Add a **Hacker News** node and connect it to the **Start** node.
2. Select the **All** resource for the **Hacker News** node. Now, add an **Execute Workflow** node and connect it to the **Hacker News** node. We'll refer to this workflow as **Workflow 1** from here on.
3. Save this **Workflow**. It should now look something like this:

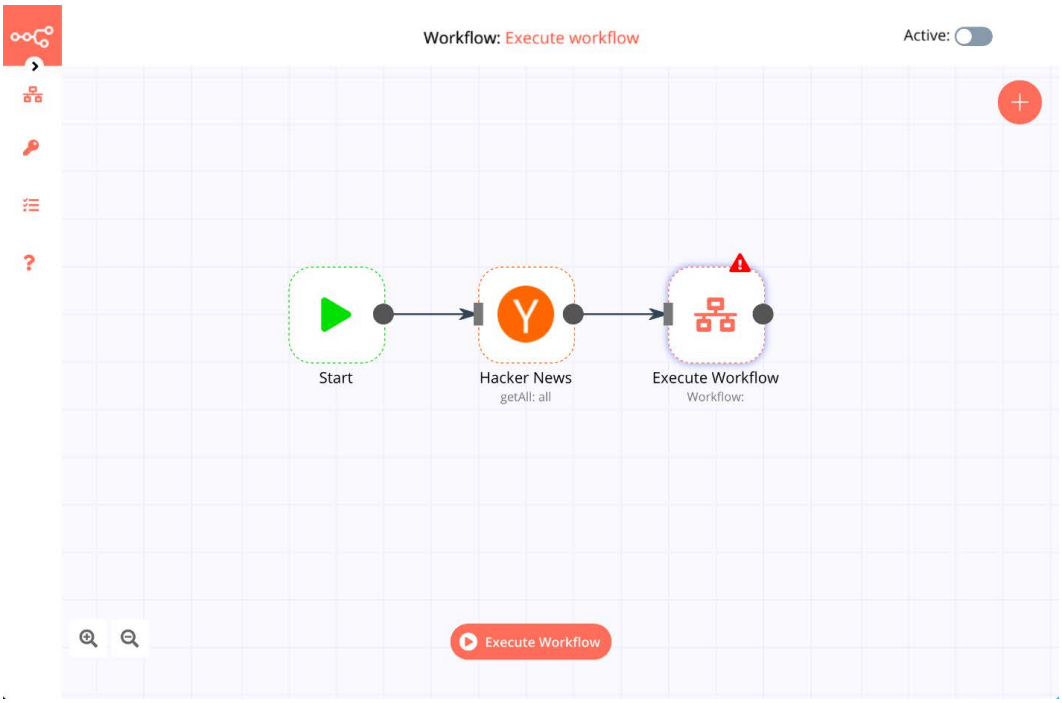

Figure 7.1 – A workflow that was created using the Execute Workflow node

4. In a new tab, create a new **Workflow**. In this new **Workflow**, add a **Set** node and an **Airtable** node.
5. Connect the **Set** node to the **Start** node and the **Airtable** node to the **Set** node. Set it up so that only the `title` and `url` properties of the article get set and inserted into the **Airtable** node. Since we don't have any data in our **Workflow**, using expressions might be a bit tricky. You can use the following expressions here:
   - `{{$json["title"]}}`
   - `{{$json["url"]}}`

6. Now, configure the **Airtable** node, save this **Workflow**, and obtain its ID. We'll refer to this workflow as **Workflow 2** from here on. You can find the ID of a **Workflow** using its URL (as shown in *Figure 7.2*). For example, the URL of my saved **Workflow** is `http://localhost:5678/workflow/297`, so the ID would be **297**.

   Here's what my **Workflow** and its ID look like:

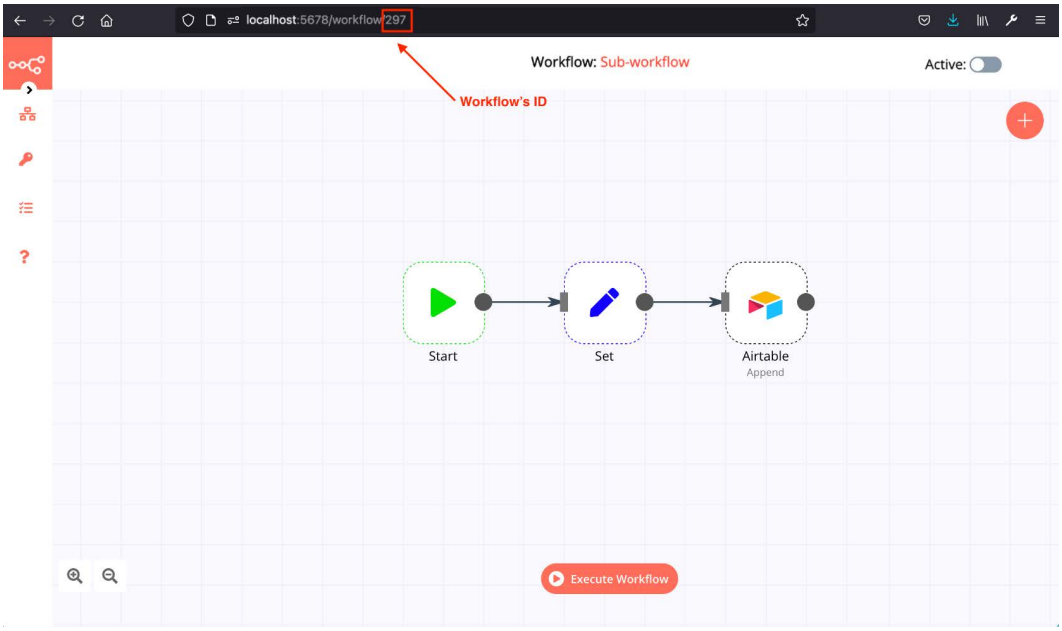

Figure 7.2 – A workflow for filtering and inserting data into Airtable

7. Now, go back to the workflow with the **Execute Workflow** node and enter the ID of the new workflow. Execute the workflow.

   You will notice that **Workflow 1** runs **Workflow 2**, gets the data that's returned by it, and displays it in the output of the **Execute Workflow** node. This is how you can share data between multiple workflows in n8n and break them into more manageable chunks.

Let's understand how the data passes between the two workflows that we created. The **Execute Workflow** node in **Workflow 1** passes the data to the **Start** node of **Workflow 2**. Because of this, all the nodes of **Workflow 2** must be connected to the **Start** node. The last node of **Workflow 2** sends the data back to the **Execute Workflow** node in **Workflow 1**, as shown here:

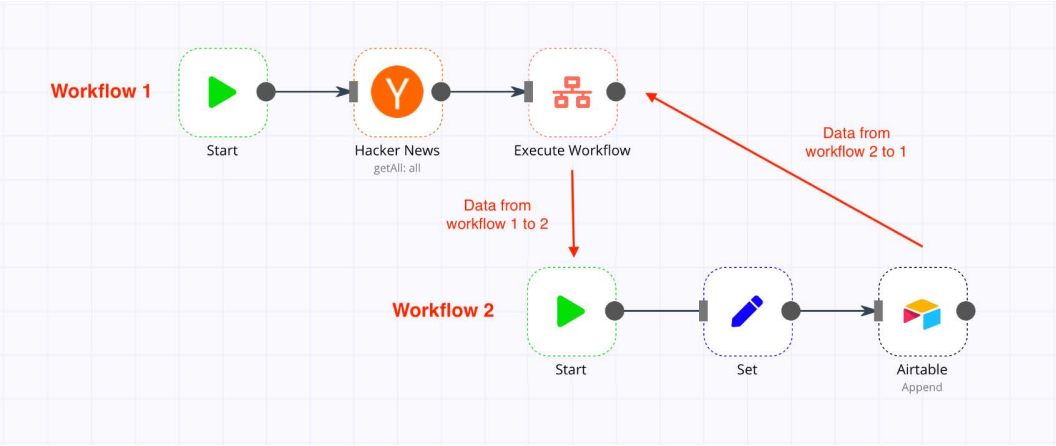

Figure 7.3 – Flow of data between the two workflows

Let's take a look at the options provided by the **Execute Workflow** node. The **Source** field allows you to specify how the node should look for the workflow:

- **Database**: Loads the workflow from the database by its ID.
- **Local File**: Loads the workflow from a locally saved file. This path must be relative to where n8n is running.
- **Parameter**: Loads the workflow from a parameter. Here, you can provide the workflow JSON.
- **URL**: Loads the workflow from a URL.

Now, let's look at how we can merge datasets in an n8n workflow using the **Merge** node.

Merging datasets

In *Chapter 6*, *Powering Your API with a No Code Database*, we learned how to use Airtable as a no-code database for our application. Let's build on that example to visualize what a database for a newsletter management app could look like. Typically, databases have different tables for different categories of data points, and we can reference data between different tables using unique IDs. Let's understand this with the help of an example:

1. Open the Airtable base called **The n8n book** that you created in *Chapter 6*, *Powering Your API with a No Code Database*. Rename **Table 1** to **Users**.
2. Add two new tables to it called **Newsletters** and **Stats**. In the **Newsletters** table, add the following columns:

   NewsletterID (single-line text)

   Subject (single-line text)

   Content (long text)

   Clicks (single-line text)
3. Add an entry to the table called **NewsletterID1**. It should now look like this:

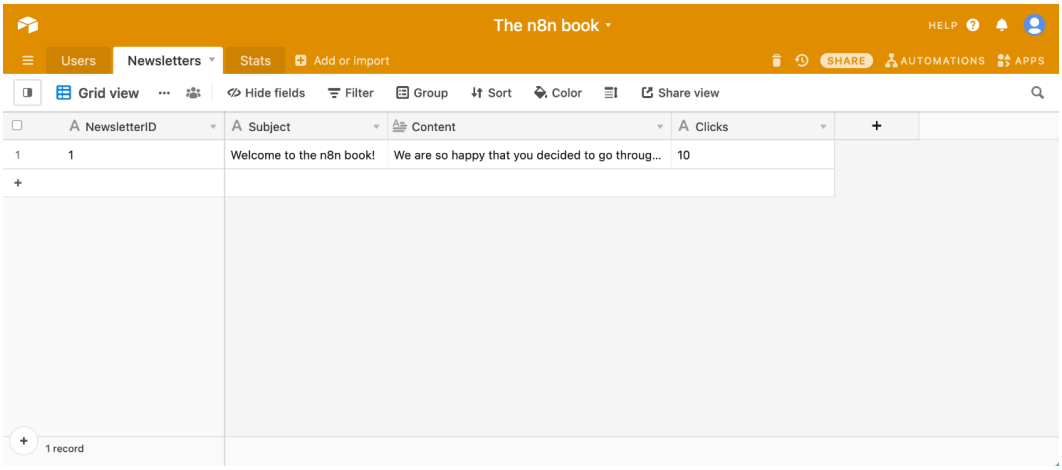

Figure 7.4 – Creating and filling the Newsletters table

4. In the **Stats** table, add the following columns:

   UserID (single-line text)

   Clicks (single-line text)

5. Add an entry to the table called **UserID1**. It should now look like this:

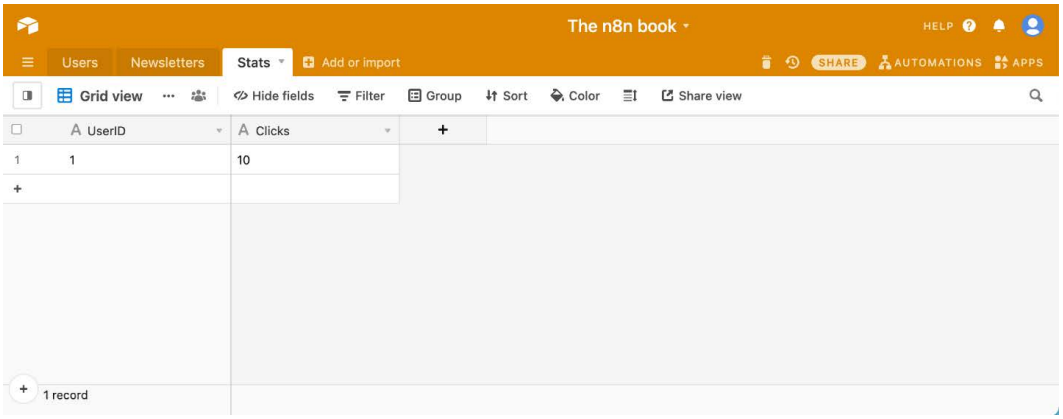

Figure 7.5 – Creating and filling the Stats table

Now, we have three tables with unique IDs that can be used to cross-reference data between the different tables.

Let's say that we want to know how many clicks have been made by each user. We want to know this number so that we can send them an email, thanking them for their active engagement if it is greater than 9. To build out a workflow for this, we'd need to have two crucial pieces of information: the user's email address and their number of clicks. Both these details reside in different tables that are linked only by the unique **UserID**. Let's use the **Merge** node to get all the information that we need.

1. Open the Editor UI and add an **Airtable** node. Configure it so that it lists all the data from the **Users** table. The details that we need are nested in the `fields` object. Since we don't need the timestamp and the **Airtable** node's ID, we can get rid of them. You can use either the **Set** node or the **Function**/**Function Item** nodes to do that. I used the **Function** node with the following code:

```
const newItems = [];

for(let i=0; i<items.length; i++) {
newItems.push({json: items[i].json.fields});
}

return newItems;
```

The preceding code ensures that only the `fields` array is returned by this node. The data should now look as follows:

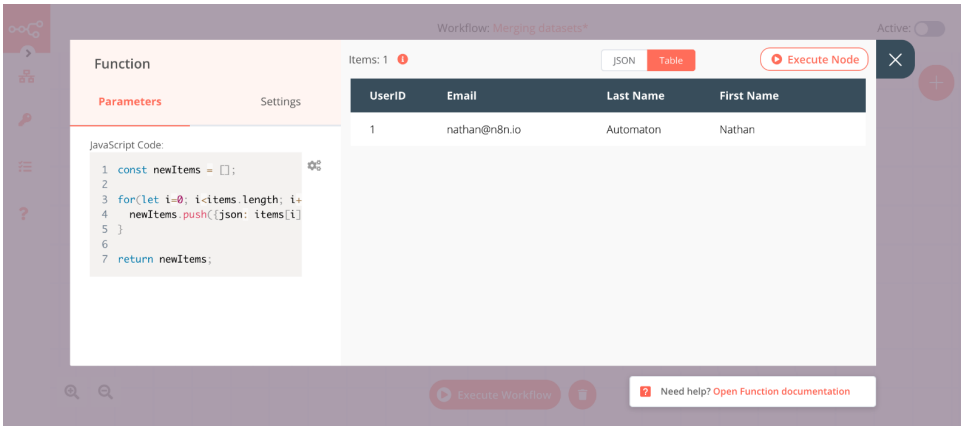

Figure 7.6 – This is what the data from the Function node should look like

2. Now, add a **Merge** node and connect **Input 1** to the **Function** node.
3. Now, we need to get the data from the **Stats** table. Perform the same steps that we mentioned at the beginning of this section by adding an **Airtable** node (which we will call **Airtable1**), along with a **Function** (or **Set**) node (which we will call **Function1**), and connecting it to **Input 2** of the **Merge** node. The workflow should look like this:

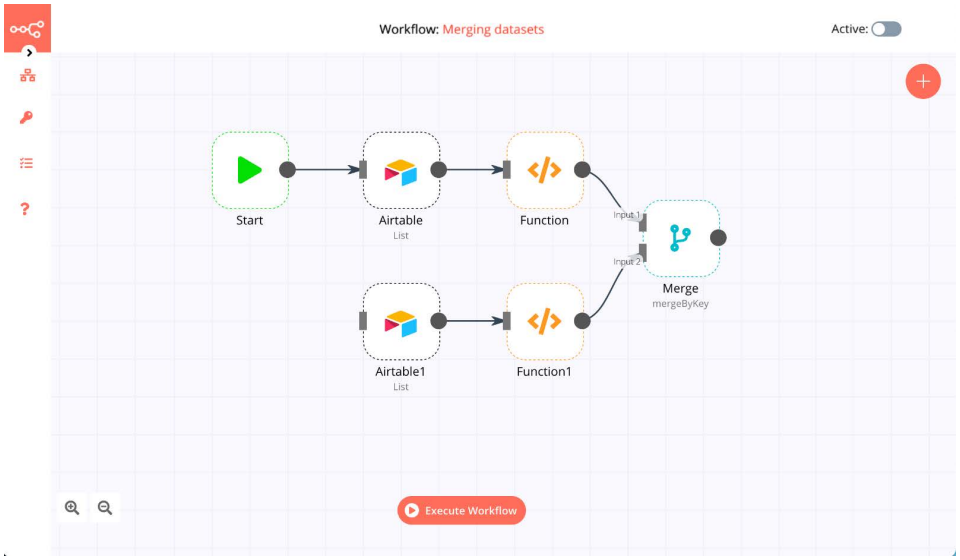

Figure 7.7 – This is what the workflow should look like

4. Open the **Merge** node and set **Mode** to **Merge By Key**.
5. Enter **UserID** in the **Property Input 1** and **Property Input 2** fields. We are doing this since **UserID** is the field linking the two datasets from the two different tables together.
6. Execute the workflow; the result of the **Merge** node should look something like this:

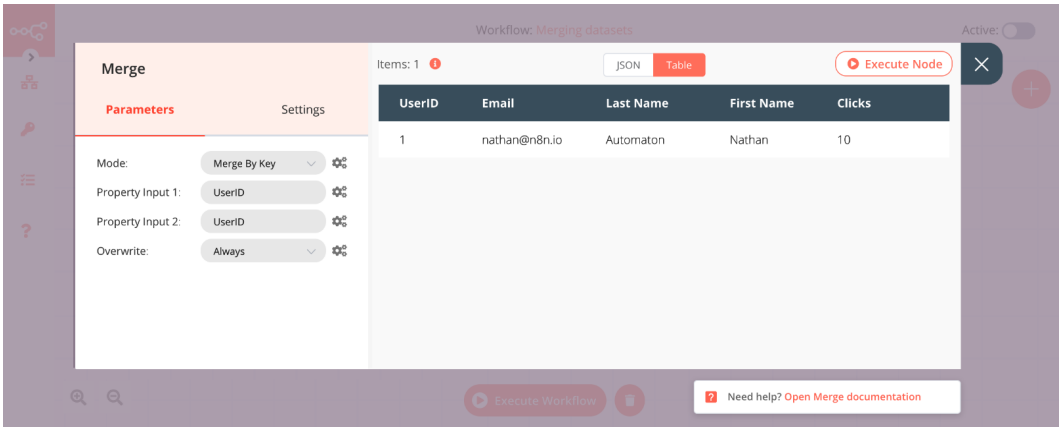

Figure 7.8 – The Merge node merging the data from two different tables

> **Note**
> The latest version of n8n also has an Item Lists node that can be used.

Now that we have the two pieces of data that we need, we can add an **IF** node and an email node (such as the **Send Email** or **Gmail** node) after the **Merge** node so that we can thank the engaged readers of the newsletter.

The **Merge** node has several different modes that can be used to merge the data in the format that works best for your use case. Now that we know how we can merge datasets inside a workflow using n8n, let's learn how to perform calculations and analytics in n8n using JavaScript.

Performing calculations and analytics

You can use JavaScript in n8n within expressions and use the **Function** nodes to perform mathematical calculations and basic analytics.

Let's use the newsletter database from the *Merging datasets* section to try out some calculations. We have added a few more records to the tables. You can clone the **Airtable** using the following link if you'd like to use our records: `https://airtable.com/invite/l?inviteId=invRJMGCMu7HWQzKW&inviteToken=3b6fbc536ccc17cf24fbeb01b5e8a253fe99afd27616f3abaeaffb046cedf8aa&utm_source=email`.

Let's calculate a few things from our **Airtable** database:

- The number of users
- The average number of clicks per newsletter
- The highest number of clicks by a user

To do this, follow these steps:

1. Open the **Editor UI** and open a new **Workflow**. Add an **Airtable** node and list all the records from the **Users** table.
2. Add a **Set** node and connect it to the **Airtable** node. Our **Workflow** should look like this:

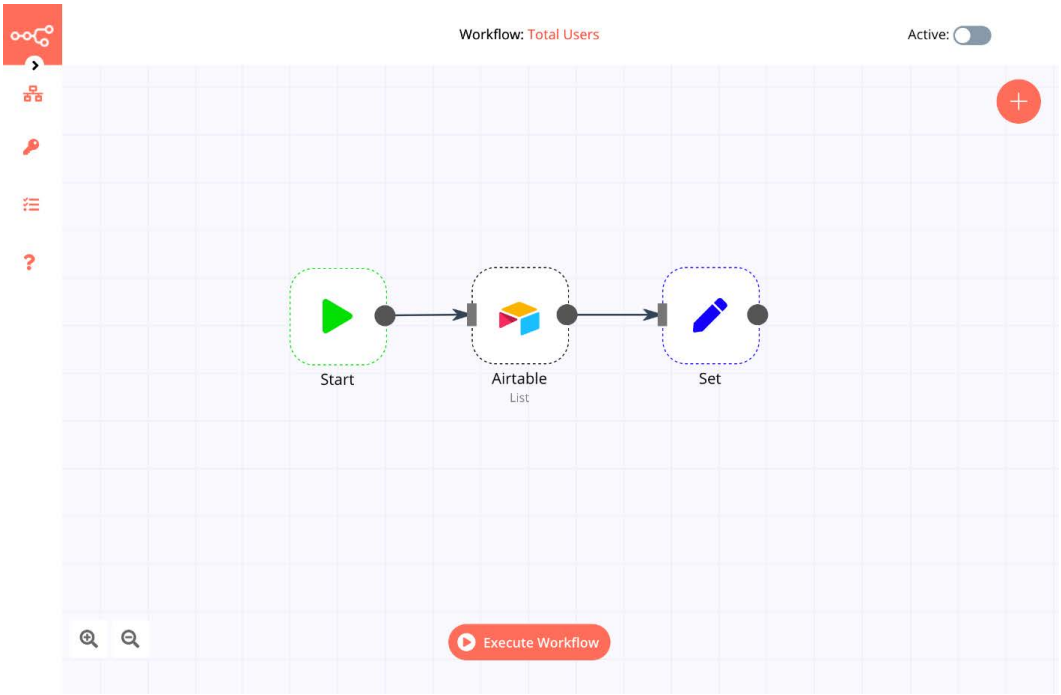

Figure 7.9 – A workflow for calculating the total number of users in the Users table

3. Set **Keep Only Set** to *true*. This removes all incoming workflow data and only appends the new values that have been configured in the **Set** node.
4. Add a value of the **Number** type and name it **Total Users**. Add an expression for the **Value** field:

```
{{$items.length}}
```

   This will calculate the total number of items that are returned by the **Airtable** node, which is also the total number of users. Executing this node will cause this value to be returned three times (once for each item).
5. In the node's **Settings** area, set **Execute Once** to *true*. Your workflow should now calculate the total number of users in the **Users** table for you. The result from the **Set** node should look like this:

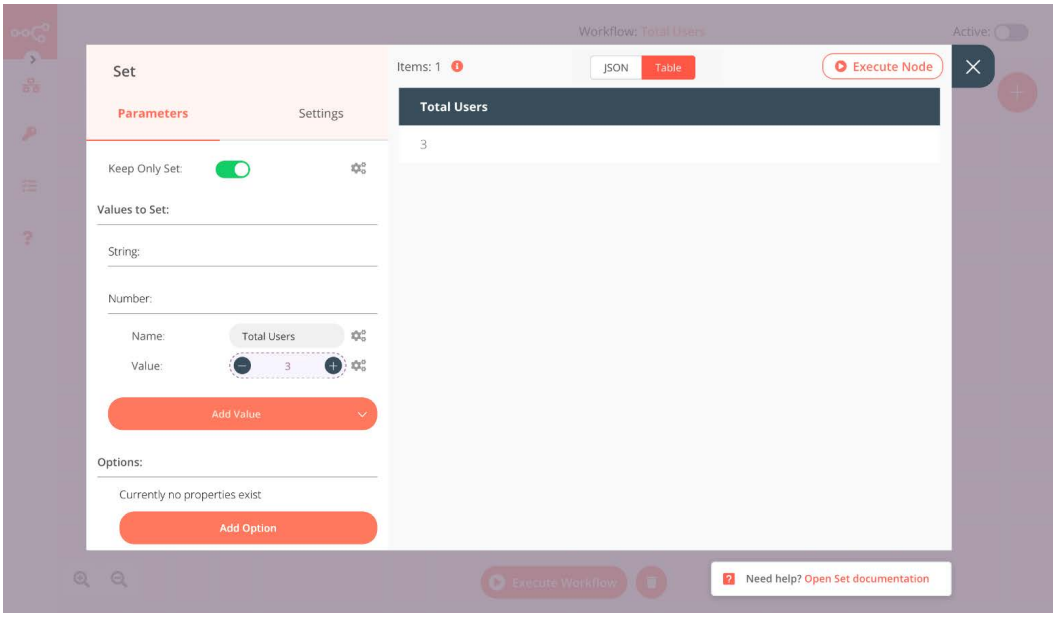

Figure 7.10 – Output of the Set node

Let's calculate the **Average clicks** value per newsletter.

1. Create another **Workflow** and add an **Airtable** node that lists all the entries from the **Newsletters** table. Add a **Function** node and connect it to the **Airtable** node. The workflow should look like this:

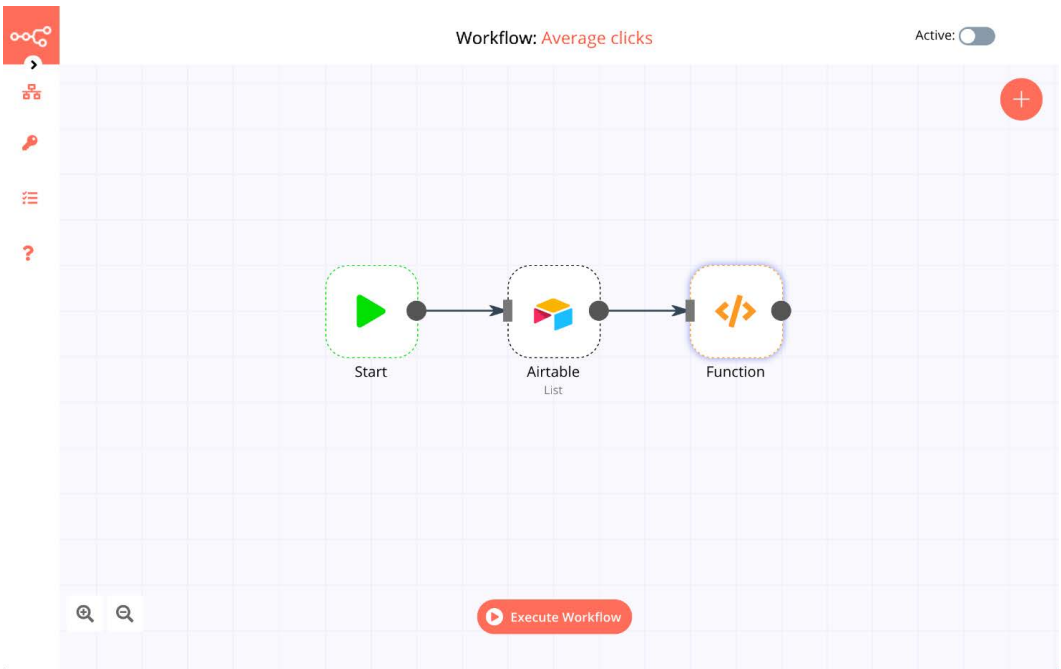

Figure 7.11 – A workflow for calculating the average clicks per newsletter

2. In the **Function** node, add the following JavaScript code:

```
let total = 0;
let average = 0;

for (let i=0; i<items.length; i++) {
  total = total + parseInt(items[i].json.fields.Clicks);
}

average = total/items.length;

return [{json: {average_clicks: average}}];
```

In the preceding code, we iterated over all the records that were returned by the **Airtable** node and added them to the `total` variable. We used the `parseInt()` function because the value of `Clicks` is of the `String` data type and we need to convert it into the `integer` data type. Finally, we calculated the `average` value by dividing the `total` clicks by the number of newsletters, which we calculated with `items.length` (exactly like we did in the preceding workflow). This provides us with the average clicks per newsletter. The following screenshot illustrates this:

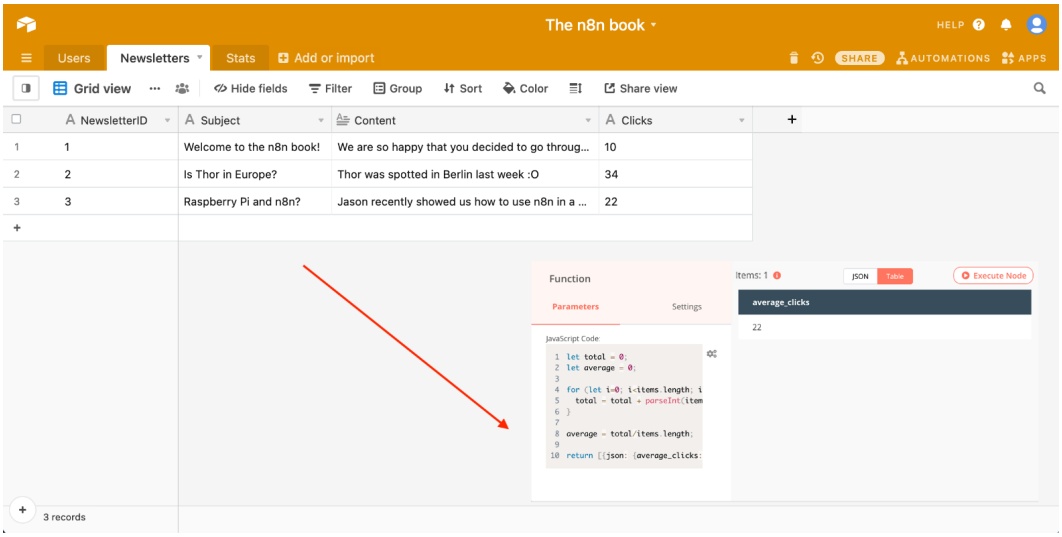

Figure 7.12 – Calculating the average clicks per newsletter

Finally, let's calculate the highest number of clicks by a user.

3. Create another workflow and add an **Airtable** node that lists all the entries from the **Stats** table. Add a **Function** node and connect it to the **Airtable** node. The workflow should look like the one shown in *Figure 7.11*.
4. In the **Function** node, add the following JavaScript code:

```
const clicks = [];
let highest = 0;

for (let i=0; i<items.length; i++) {
clicks.push(items[i].json.fields.Clicks);
}
```

```
highest = Math.max(...clicks);

return [{json: {highest_clicks: highest}}];
```

In the preceding code, we added all the number clicks to an array called `clicks`. Then, we used the `Math.max()` function to find the maximum value in that array. This provides us with the highest number of clicks per user. The following screenshot illustrates this:

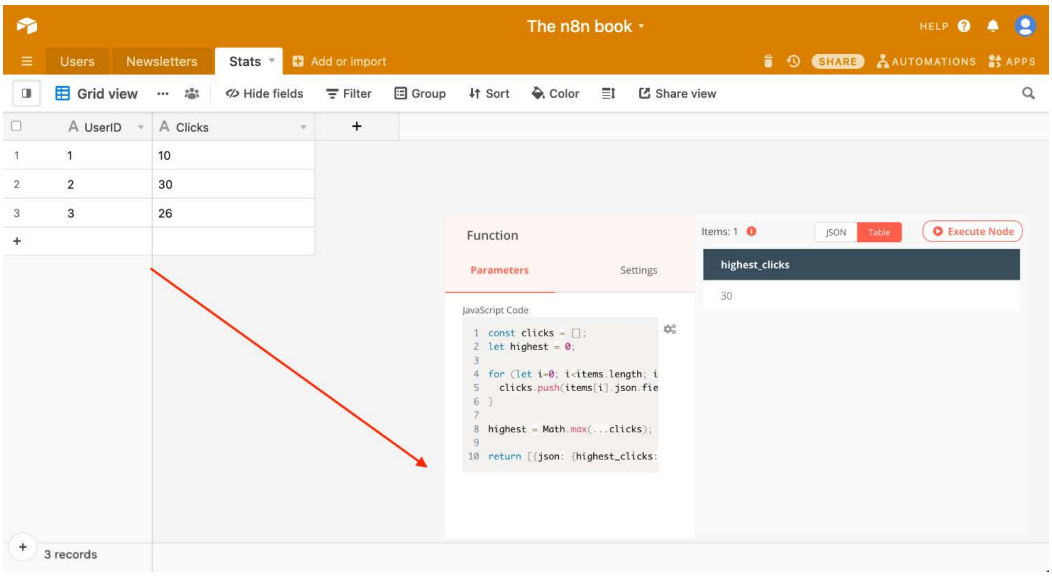

Figure 7.13 – Calculating the highest number of clicks per user

`Math` is a useful built-in object that can be utilized for a lot of these calculations. You can find out more at `https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Math`.

These were some basic examples of how you can use JavaScript to perform calculations and create workflows for analytics to gain insights from the data that you accrue through your APIs and apps.

Summary

In this chapter, we learned about sharing data between multiple workflows in n8n, merging data coming from different sources within a workflow, and using JavaScript to perform calculations and analytics from within a workflow. The concepts we learned in this chapter will help you when you're sending data to other services or no-code tools using our custom API.

In the next chapter, we are going to introduce the Bubble APIs and integrate them into n8n. We will learn how to work with Bubble data and workflows, along with how to configure n8n to receive events and data initiated by Bubble.

8
Utilizing the Bubble API in n8n

Regardless of how hard we try, we can't be good at everything! Some people are great organizers, others are fantastic planners, while still others excel at public speaking. When we notice that we are good at something, we tend to focus on that thing and pursue it, making us even better at it.

No-code tools are like that as well. They discover what they are really good at and focus on it. n8n is really good at connecting systems and automating their tasks, which is the major focus of this tool.

And sometimes, just as with people, if no-code tools want to get something accomplished but need a little help, they will turn to a friend.

For us, we have turned to Bubble to be n8n's partner. Bubble is designed to be a really good web app development tool, allowing us to create a web frontend for n8n. It allows no-code builders to design web pages for submitting information to n8n or even complete web applications so that n8n can then connect the data to other systems and automate the processes.

This chapter covers the following topics:

- Introducing the Bubble **application programming interface** (**API**)
- Understanding Bubble's data structure
- Understanding Bubble's workflow engine
- Using Bubble's Data API
- Using Bubble's Workflow API
- Receiving events and data from Bubble

Once you have completed this chapter, you will know how to do the following:

- Communicate between Bubble and n8n.
- Access Bubble's data using the Data API.
- Use Bubble's workflows and interact with them using the Workflow API.
- Receive events and data from Bubble in n8n.

Technical requirements

You can find the completed code examples for the chapter on GitHub at `https://github.com/PacktPublishing/Rapid-Product-Development-with-n8n/tree/main/Chapter%208`.

Introducing the Bubble API

If you have worked with other APIs in the past, you should feel entirely comfortable working with the Bubble API. It follows relatively standard API design practices, such as using regular **HyperText Transfer Protocol** (**HTTP**) responses and **HTTP Secure** (**HTTPS**) for over-the-wire encryption.

However, some specifics about the API will make it easier for you to find success early if you learn them upfront.

Bubble API endpoints

The primary endpoint for your app depends on a few factors, such as the following:

- Are you using a custom domain?
- Are you using the live or development environment?
- Which version of the API are you using?

Base URL

The first item to look at is whether you are using a custom domain or not for your Bubble app. This is used to figure out your base **Uniform Resource Locator** (**URL**). You can determine whether or not you are using a custom domain by opening up your Bubble app, going to the **Settings** menu, and then clicking on the **Domain / email** tab, as illustrated in the following screenshot:

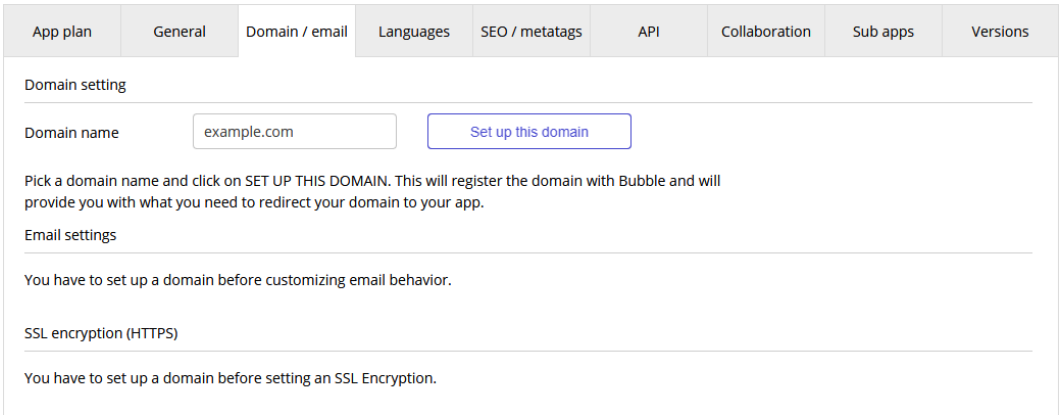

Figure 8.1 – Bubble domain settings without a custom domain

If you have a custom domain, it is registered in this location. Otherwise, it displays the default settings, as shown in *Figure 8.1*.

For default domain settings, the base URL is `https://<appname>.bubbleapps.io`, where `<appname>` is the name you have provided for your app. For example, if you named your application `fancyApp`, then the base URL is `https://fancyApp.bubbleapps.io`.

If you have a custom domain for your app, then the URL is your custom domain.

Live or development

The next factor used to determine your API endpoint is whether you are using the live or development version of your Bubble application. By default, you will be in the development version. Once you are ready to make your application public, you can then switch it to the live version.

Be aware that the API endpoints change when you go from the development version to the live version.

If you use the live version of the application, then the path after the base URL is `/api`, but if you are performing some testing on the development version of your application, then this subdirectory is `/version-test/api`.

API version

Chances are, you are using the latest version of the API. Version 1.1 was released on January 19, 2017, and if you created your application after January 19, 2017, you are probably using version 1.1.

The path after specifying your environment is your API version. This is either `/1.0` or `/1.1`, based on when your application was created.

Workflow or data

Bubble can be broken into two different pieces: workflow and data. Workflow deals with events that occur within Bubble (for example, a button is clicked), and then it performs specific actions based on how you have configured Bubble.

The data piece refers to the built-in database that Bubble uses (for example, information about a newsletter). It is all about saving, changing, and deleting data within the Bubble environment.

Once your API version is specified, you need to indicate whether you access the Workflow or Data API. This is done in the next part of the path. If you are accessing the Workflow data, the next part of the path is `/wf`. Meanwhile, if you want to access the Data portion of the API, this part of the path is `/obj`.

Name

The final part of the endpoint is to specify the name of the item you are accessing. Depending on whether you are accessing a workflow or a data item, this value specifies that item.

For example, if you are accessing a data item named `users`, the path portion after `/obj` is `/users`.

Building your API endpoint

Now that we have a solid understanding of how the API endpoints are put together, let's demonstrate how to build an API endpoint using an example.

Let's assume we are accessing the development version of the newsletter information collected by the `myApp` application we created in May 2019. There is a plan to use a custom domain for this application, but it hasn't been deployed yet.

Since this application doesn't have a custom domain, the base URL is `https://myApp.bubbleapps.io`. We are accessing the development environment, so we add `/version-test/api` to the path. This application uses version 1.1 of the API as it was created after January 19, 2017, and this portion of the path is `/1.1`. We then add `/obj` to the path followed by the `/newsletter` data item name to access the Data API.

Our final API endpoint in this example is `https://myApp.bubbleapps.io/version-test/api/1.1/obj/newsletter`.

Bubble API settings

You can control several aspects of the Bubble API for your application. By going to **Settings** in the application and selecting the **API** tab, as illustrated in the following screenshot, you can make several changes:

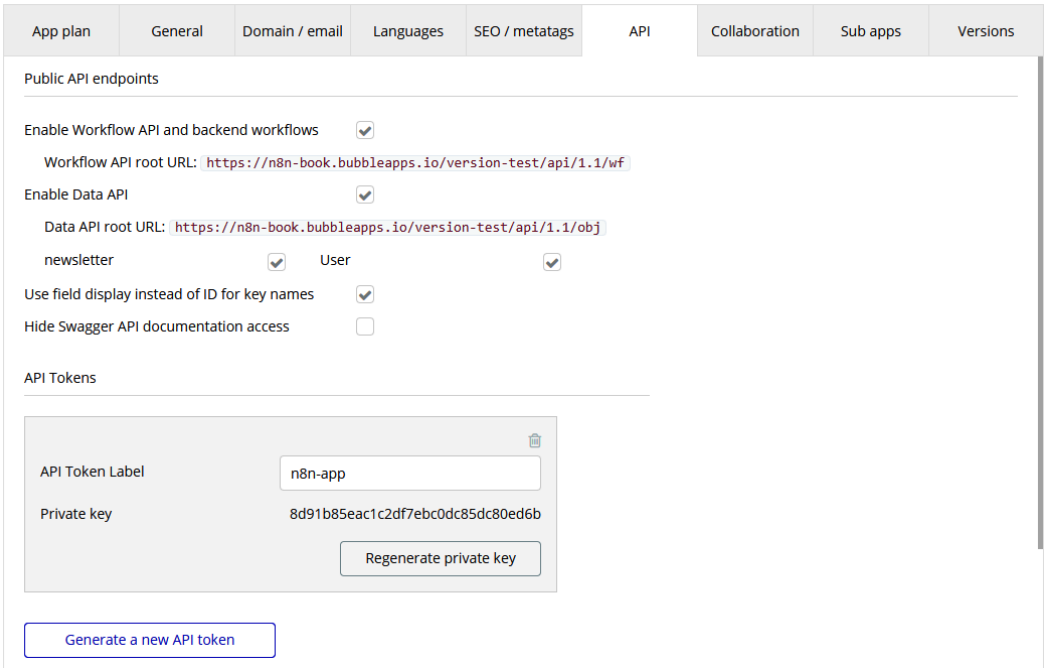

Figure 8.2 – Bubble API setting options

Enabling/disabling API access

You can enable or disable API access by selecting or deselecting the checkbox next to **Enable Workflow API and backend workflows** or **Enable Data API**.

You can also specify workflows or data items by selecting or deselecting the checkbox next to their name.

API tokens

To secure your API access, it is essential to create a private API key. You can regenerate an API token in this section or create a new token altogether.

You can authenticate using the API token by adding a header with a key of **Authorization** and a value of `Bearer <API key>`, as illustrated in the following screenshot:

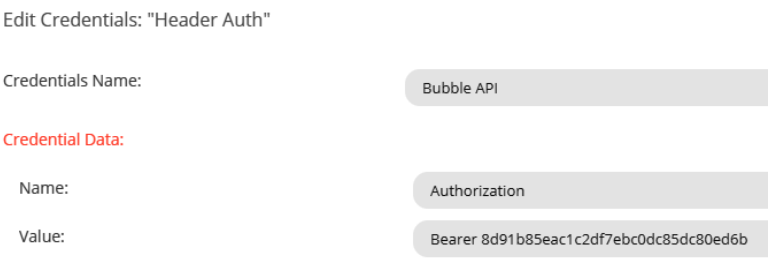

Figure 8.3 – Bubble API header authentication

Now that we have a better understanding of how Bubble's API works so that we can access data from other systems, let's delve a bit deeper into the actual structure of that data. This is important so that you know exactly what you are retrieving when using the API and you will be able to properly navigate around the API to retrieve specific data.

Understanding Bubble's data structure

To properly use the Bubble API to access and modify data, it is crucial to understand how data is stored in Bubble.

Data types

In Bubble, a data type is essentially the equivalent of a **JavaScript Object Notation** (**JSON**) object. It is defined in Bubble under the **Data** section for the application and provides a list of fields (represented by a key in JSON). Each field is a property of the data type. You can see a sample Bubble data type in the following screenshot:

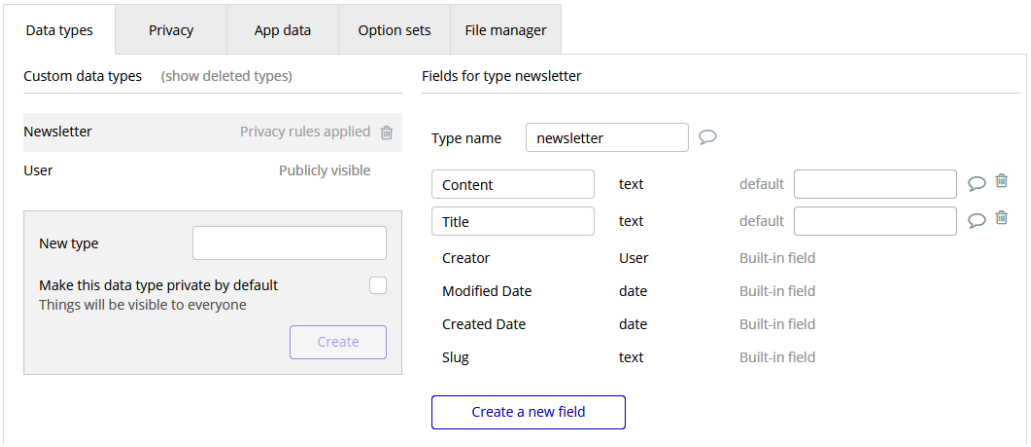

Figure 8.4 – Sample Bubble data type

For example, the **Newsletter** data type in the preceding screenshot has six fields, two of which are custom, as follows:

- **Content** —The actual body of the newsletter
- **Title** —The newsletter's title

Four of the fields are default fields, as follows:

- **Creator**—The username of the person who created the record/newsletter entry
- **Modified Date**—The date the record was last created, changed, or updated
- **Created Date**—The date the record was initially created
- **Slug**—Shortcut used for accessing the record, usually when using a URL form

Data security (privacy)

There are two general privacy or data security settings for data types, as follows:

- **Public**
- **Private**

The **Public** privacy setting gives everyone access to read the data in the data type. This is generally frowned upon unless the data is truly public and you do not care who can read the information.

The **Private** privacy setting prevents anyone but the creator of the data type from actually interacting with the data. This is the preferred security setting for data types.

The following screenshot shows Bubble private permissions with API access:

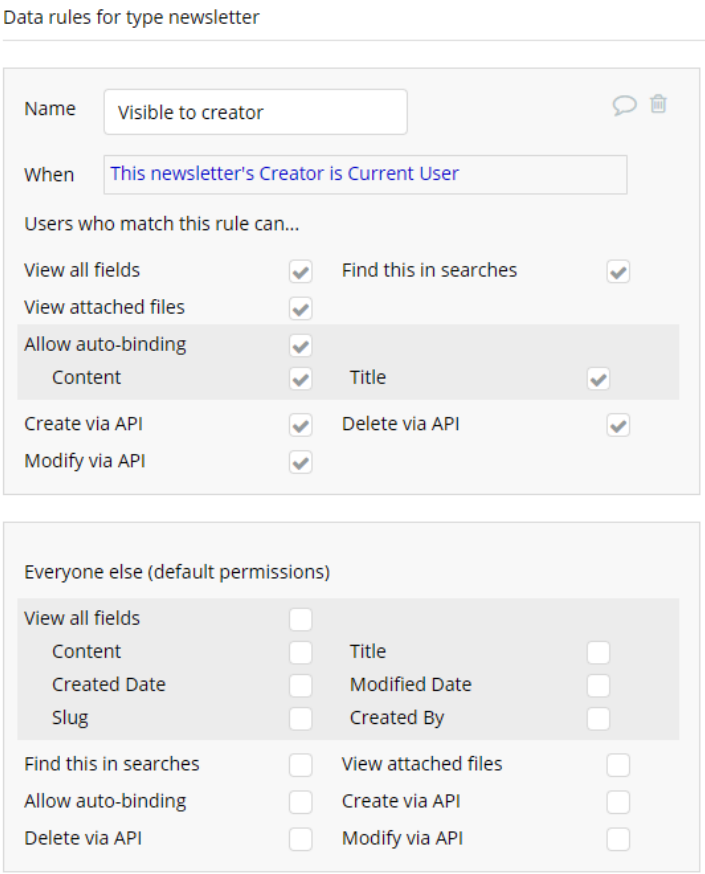

Figure 8.5 – Bubble private permissions with API access

To access secured data via the API, you need to do a few things, as follows:

1. Enable access to the data type in the **Settings API** tab for the application.
2. Select at least one of the API settings (**Modify by API**, **Delete via API**, and **Create via API**) in the **Privacy** tab for the data type in the **Data** section of the application.
3. Authenticate using the API key from the **Settings API** tab for the application.

Understanding Bubble's data structure is a crucial step to enabling automation. While n8n is our preferred tool for automation, Bubble has its own workflow engine built into the system, which we will summarize next.

Understanding Bubble's workflow engine

Bubble has its own version of automation built into the system. It is not as robust or functional as n8n, as it is designed to automate and communicate with itself and excels at this. Each workflow executes one or more steps to perform actions supported by the Bubble application.

There are two different kinds of workflows in the Bubble environment. The first kind is frontend workflows. These are workflows designed to interact with users when they perform an action on one of Bubble's application pages. For example, when you click on the **Submit** button in Bubble, this executes a frontend workflow.

Similarly, there are also backend workflows. These workflows are designed to be non-interactive with the user and perform automated tasks based on various triggers such as time of day, changes to data, or input/requests from an API.

When interacting with Bubble's workflow engine, n8n specifically talks with backend API workflows.

Depending on how the API workflow is configured, you can send data to the API, which will be used in the workflow. For example, the `send-email` workflow will accept the following inputs from the API:

- `to`
- `subject`
- `body`

This is illustrated in the following screenshot:

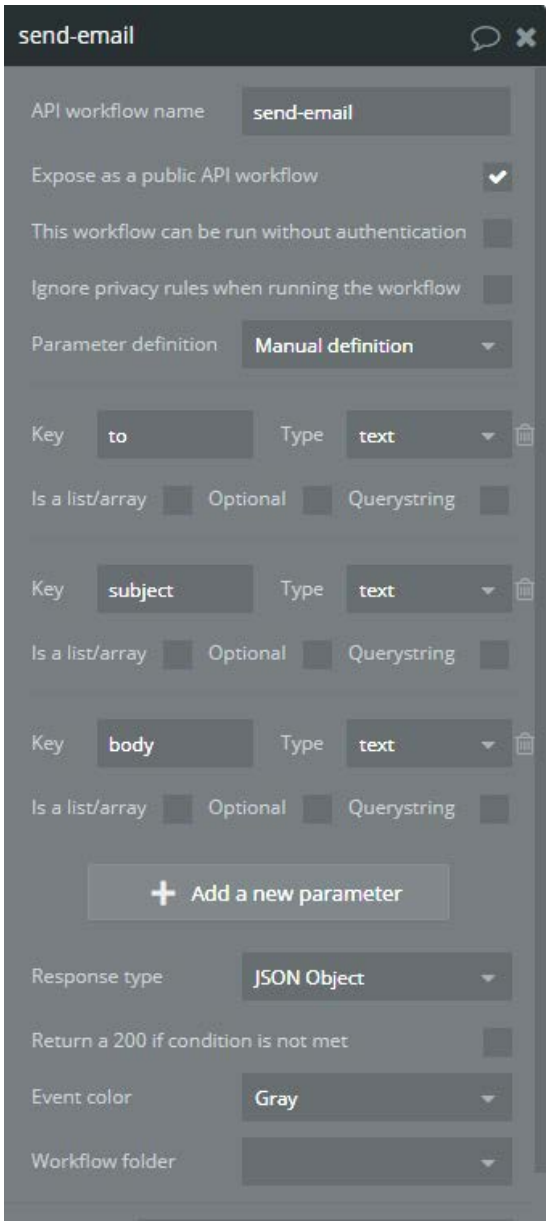

Figure 8.6 – Sample backend API workflow configuration screen

So, now that we have a general understanding of the workflow engine, let's begin using the actual Bubble APIs, starting with the Data API.

Using Bubble's Data API

Accessing Bubble's data with n8n is relatively trivial using the Data API. Depending upon the endpoint accessed and the HTTP method used, you can manipulate the data in the Bubble application in any way that you want.

To load some sample n8n nodes to work with the Data API, please see the `Bubble_API.json` workflow at `https://github.com/PacktPublishing/Rapid-Product-Development-with-n8n/blob/main/Chapter%208/Bubble_Data_API.json`.

Authentication

Before you can work with the Bubble Data API in n8n, you will need to create proper credentials. We will be using the **HTTP Request** node for all of our API interactions, so we need to create a **Header Auth** credential, as illustrated in the following screenshot:

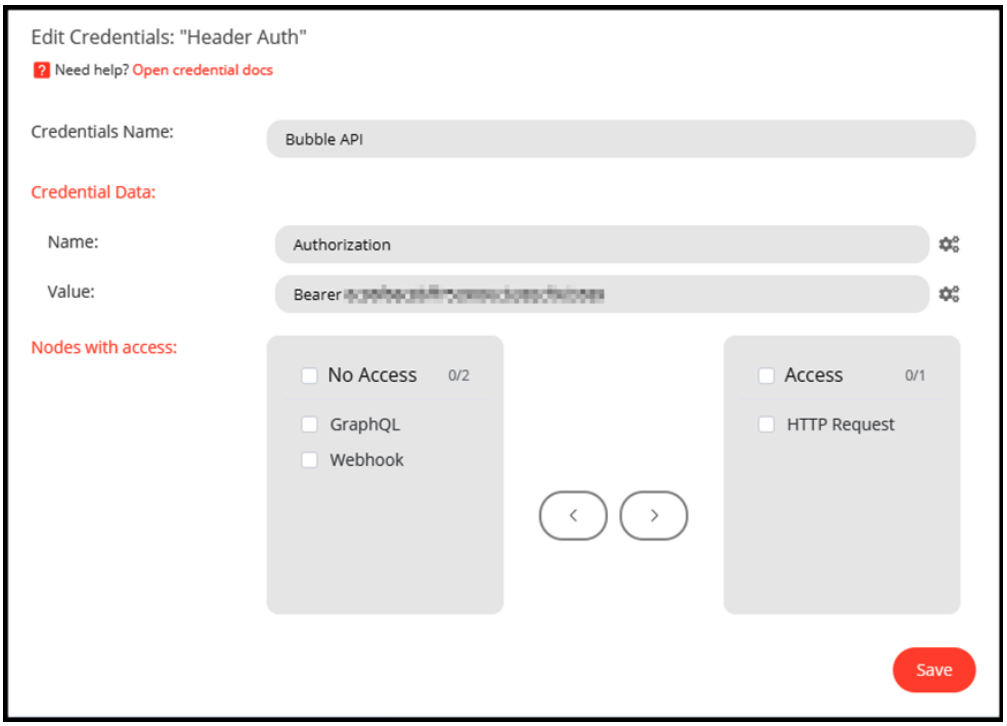

Figure 8.7 – Header Auth credential for accessing Bubble APIs

It is important that the **Name** field under **Credential Data** be populated with **Authorization**, and the **Value** field must be populated with **Bearer**, followed by your API private key, which can be found in the **Settings** section under the **API** tab.

You must also make certain that you provide access to the **HTTP Request** node.

Data manipulation

Once you have created your credentials, you can now use them to work with Bubble data.

Here is a list of the different ways in which you can retrieve and manipulate data in Bubble using the Data API:

Action	Method	Endpoint
Retrieve records	`GET`	`https://appname.bubbleapps.io/api/1.1/obj/typename`
Retrieve record	`GET`	`https://appname.bubbleapps.io/api/1.1/obj/typename/id`
Modify record	`PATCH`	`https://appname.bubbleapps.io/api/1.1/obj/typename/id`
Replace record	`PUT`	`https://appname.bubbleapps.io/api/1.1/obj/typename/id`
Delete record	`DELETE`	`https://appname.bubbleapps.io/api/1.1/obj/typename/id`
Create record	`POST`	`https://appname.bubbleapps.io/api/1.1/obj/typename`

In each of the endpoints, two special words represent actual unique values from your Bubble application, as follows:

- `typename`—This is the actual name of the data type in Bubble. This is generally the name of the data type in lowercase with all spaces removed.
- `id`—The **unique identifier** (**UID**) number for the record in Bubble.

For example, if you had a data type of **Newsletter** and the record ID was `1626690254917x27912944044325693O`, you would be able to retrieve this record using the `https://appname.bubbleapps.io/api/1.1/obj/newsletter/1626690254917x27912944044325693O` endpoint.

We now have all the information we need to start working with the data in the Bubble application.

n8n uses the **HTTP Request** node to perform all data manipulation, as illustrated in the following screenshot:

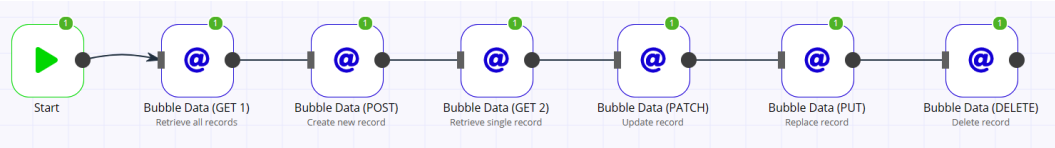

Figure 8.8 – Using the n8n HTTP Request node to manipulate Bubble data

Using the information provided in the previous table, we can configure the **HTTP Request** node to work with the data however we wish. In the following screenshot, you can see the general settings for the **HTTP Request** node:

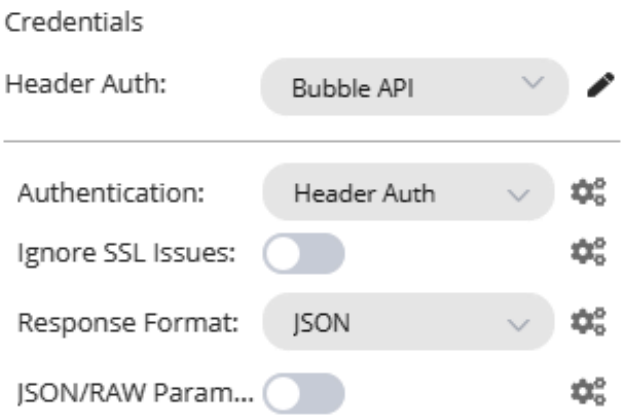

Figure 8.9 – General HTTP Request node settings

In general, all of the **HTTP Request** nodes will have the following settings in common:

- **Header Auth**—Select the Bubble API credentials that you created earlier.
- **Authentication**—**Header Auth**.
- **Ignore SSL Issues**—**False**.
- **Response Format**—**JSON**.
- **JSON/RAW Parameters**—**False**.

Once these are set, you can now go on to the custom settings for each action.

To read all data from the API, set the **Request Method** field to `GET` and the URL to the endpoint of the data type (for example, `https://appname.bubbleapps.io/api/1.1/obj/typename`). This will generate a JSON object containing an array of all the records in the data type. The output will look something like this:

```
[
  {
    "response": {
      "cursor": 0,
      "results": [
        {
          "content_text": "Welcome to the n8n Newsletter!",
          "Modified Date": "2021-07-25T13:00:35.174Z",
          "title_text": "First n8n Newsletter",
          "_id": "1627217777235x67000748217551600O"
        }
      ],
      "remaining": 0,
      "count": 1
    }
  }
]
```

This will return up to 100 records at a time.

If you want to just get the information from a specific record, you can specify the ID by appending it to the end of the URL (for example, `https://appname.bubbleapps.io/api/1.1/obj/typename/1627217777235x67000748217551600O`). This will only return a single record.

To create a new record, set the **Request Method** field to `POST` and the URL to the endpoint of the data type (for example, `https://appname.bubbleapps.io/api/1.1/obj/typename`). Then, create a body parameter for each custom field along with the value you would like the field value to be. The process is illustrated in the following screenshot:

Figure 8.10 – Body parameters for creating, updating, or replacing record data

The API will return a status value and the ID of the new record in JSON format.

You can also update a specific record in the data type by setting the **Request Method** field to `PATCH` and then submitting the request to the record's endpoint (for example, `https://appname.bubbleapps.io/api/1.1/obj/typename/1627217777235x670007482175516000`), along with one or more body parameters with the field information that needs to be updated. If this update is successful, it will not return any data and will respond with a `204` status code.

Replacing an entire record is similar to updating a record with two exceptions. First, the value in the **Request Method** field is `PUT`, and second, the body must contain all custom field values. It also returns no data and a `204` reply code when successful.

The last action that you can do with the API is delete a record. Much as with returning a single record, you use the record's endpoint (for example, `https://appname.bubbleapps.io/api/1.1/obj/typename/1627217777235x670007482175516000`) and set the **Request Method** field to `DELETE`.

Searching for data

Sometimes, there are too many records to work through when requesting data using the `GET` request method when you don't know the record ID. Fortunately, there are a number of additional search options that will allow you to reduce the number of records returned in each API request.

The search criteria are entered as query parameters in the **HTTP Request** node as name-value pairs, as illustrated in the following screenshot:

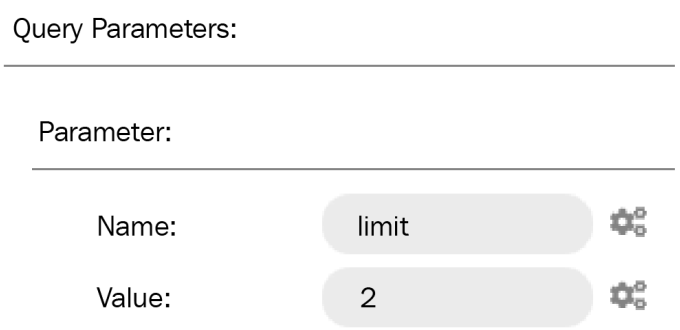

Figure 8.11 – Setting search limits using query parameters

The first way of reducing the number of records returned is to use the `limit` parameter. For example, setting the `limit` parameter to `2` will only return one record at a time.

This is all fine and good if the record you want is in the first set of records (referred to as a **page**), but what if it is on a different page? This is where the `cursor` parameter comes into play. The **cursor** is the record index that you are presently accessing. By adding the `cursor` parameter, you can start your search from a specific record. The record index counts up by one and starts at zero.

Setting the `cursor` parameter can be useful when you determine its value by using the three key/value pairs returned by a previous `GET` query. These three pairs are described here:

- `cursor`—The record index of the first item in the results
- `remaining`—The number of records that are remaining after this page
- `count`—The number of items returned

You can also control the order in which the data is returned by using the `sort_field` parameter and setting it to the name of one of the data fields. Adding the `descending` parameter and setting it to either `true` or `false` will also control whether the data is sorted in descending or ascending order.

If you set the `sort_field` parameter to `_random_sorting`, it will return the records in random order. So, to receive a single random record, you can set the `limit` parameter to `1` and the `sort_field` parameter to `_random_sorting`!

Another way to limit the number of records that are returned from the API is to use the `contraints` parameter. This contains an array of one or more JSON objects that tells the API how to limit the incoming data. Each JSON object has three key-value pairs, as follows:

- `key`—The name of the field to use for the constraint. You can use the `_all` keyword to check all fields.
- `constraint_type`—The type of constraint to apply, which can be one of the following:
  - `equals`
  - `not equal`
  - `is_empty`
  - `is_not_empty`
  - `text contains`
  - `not text contains`
  - `greater than`
  - `less than`
  - `in`
  - `not in`
  - `contains`
  - `not contains`
  - `empty`
  - `not empty`
  - `geographic_search`
- `value`—The value that is used to constrain the search.

For example, if we wanted to only return records where the `title_text` field has the word `Newsletter` in it, we would set the `constraints` parameter to the following:

```
[
  {
    "key": "title_text",
    "constraint_type": "text contains",
    "value": "Newsletter"
  }
]
```

You can have as many JSON records as you wish in this array.

By now, you should have a good understanding of how the Bubble Data API works, so let's take a closer look at the Workflow API.

Using Bubble's Workflow API

The Workflow API is generally used to activate a Bubble workflow and sometimes to pass information to the Bubble app. You will use the same credentials and the **HTTP Request** node to activate workflows and send data.

Activating a workflow

The **HTTP Request** node uses the `POST` request method to activate the workflow and will receive a `"status": "success"` response if it runs correctly.

The **URL** field for the **HTTP Request** node will be made up of the following two parts:

- **Base URL**—This is typically `https://<appname>.bubbleapps.io/api/1.1/wf`, where `<appname>` is the name of your Bubble application.
- **Workflow name**—This is the name of your workflow.

So, for example, to activate the `send-email` workflow in my `n8n-book` app, I would use `https://n8n-book.bubbleapps.io/version-test/api/1.1/wf/send-email` for the URL.

Sending data to a workflow

In order to send data to a workflow, the workflow needs to have the parameters defined in the API workflow (see *Figure 8.6*), and then the parameters need to be defined in the action as dynamic data.

In our example, the three parameters that can be sent to the `send-email` workflow via the API are these:

- `to`
- `subject`
- `body`

This is illustrated in the following screenshot:

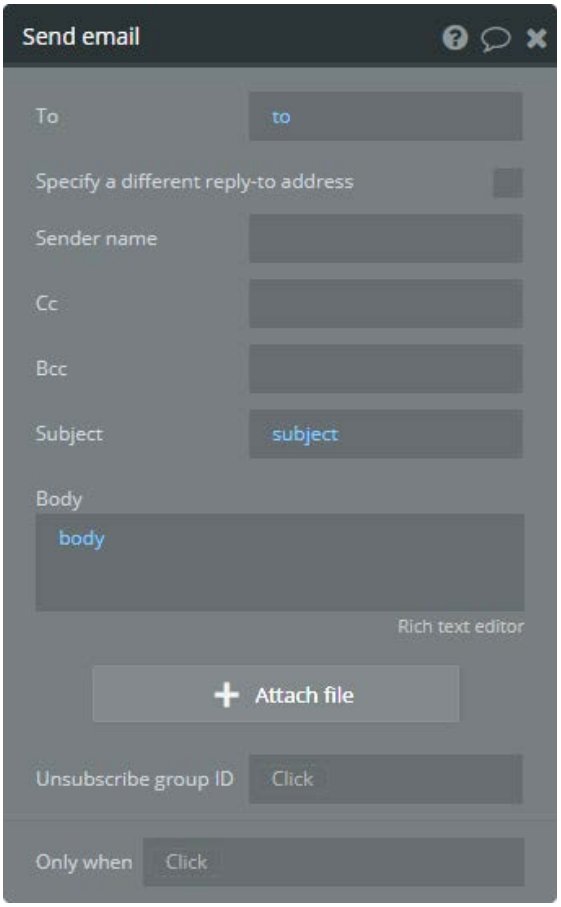

Figure 8.12 – Sample workflow action with dynamic data

To send this information to the workflow via the Workflow API using the n8n **HTTP Request** node, add a body parameter for each of the values, as illustrated in the following screenshot:

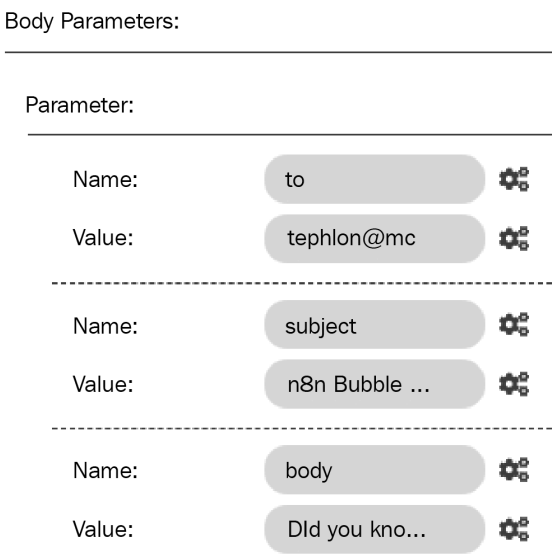

Figure 8.13 – Body parameters to send to the Workflow API

When the **HTTP Request** node is executed, it will send the information to the workflow and use it as inputs to be processed accordingly, as follows:

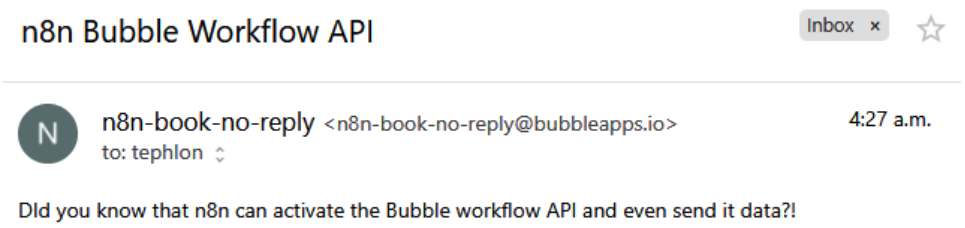

Figure 8.14 – Email sent using Bubble Workflow API and submitted data

You can get a copy of this sample workflow from the GitHub repository at `https://github.com/PacktPublishing/Rapid-Product-Development-with-n8n/blob/main/Chapter%208/Bubble_Workflow_API.json`.

Now that we know how to activate workflows and send data to Bubble, let's flip things around and get Bubble to activate workflows and send data to n8n.

Receiving events and data from Bubble

Bubble can send events to n8n and even send data. By designing n8n to receive this information, Bubble can directly execute n8n workflows and extend its capabilities.

Configuring n8n

To do this, first create a simple **Webhook** node with the following settings:

- **Authentication**—**None**
- **HTTP Method**—`POST`
- **Path**—`bubble`
- **Response Code**—`200`
- **Response Mode**—**Last Node**
- **Response Data**—**First Entry JSON**

You can also copy the test **Webhook** URL as you will need this information when configuring Bubble.

Next, add a **Set** node after the **Webhook** node and add a string value named **Response** with a value set to the following expression: `n8n received the following data on {{new Date().toDateString()}} at {{new Date().toTimeString()}}: {{$json["body"]["data"]}}`.

Now, save the workflow so that the Webhook initializes.

You can get a copy of this workflow on GitHub from `https://github.com/PacktPublishing/Rapid-Product-Development-with-n8n/blob/main/Chapter%208/Bubble_Events.json`.

Configuring Bubble

Now that n8n is ready to go, it's time to prepare Bubble.

First, make sure that the **API Connector** plugin is installed in your Bubble application. If it isn't, install it now. You can read more about the **API Connector** plugin at `https://manual.bubble.io/core-resources/bubble-made-plugins/api-connector`.

Next, we are going to configure the **API Connector** plugin. On the **API Connector** plugin page, click on the **Add another API** button. Set the **API Name** field to **n8n Workflow**.

To configure the API call, click on **expand** to open up the API call and rename the **API Call** field **Send n8n Data**. Set **Use as** to **Action**. Change the HTTP method from `GET` to `POST` and then pass the test **Webhook** URL into the field next to it.

Add a parameter to send to n8n by clicking on the **Add parameter** button. Then, for the **Key** value, enter **data**, and for **Value**, enter **Running Sloth Festival**.

The plugin should now be ready to initialize by sending information to n8n and looking at the response that it sends back. To initialize the plugin, first, go back to your n8n workflow and click the **Execute Workflow** button to start the Webhook listening for input. You should then see a screen like this:

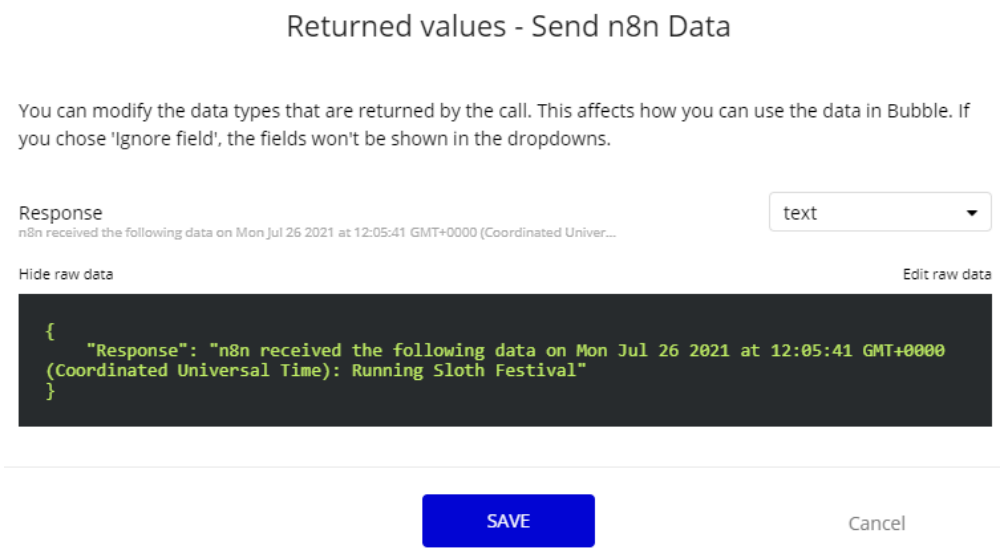

Figure 8.15 – Initializing a response from n8n

Next, return to the Bubble plugin and click the **Initialize call** button. A pop-up window should appear with the response sent by n8n. Click the **Show raw data** text to see the exact response sent by n8n. This should match the information shown in the n8n **Set** node, as illustrated in the following screenshot:

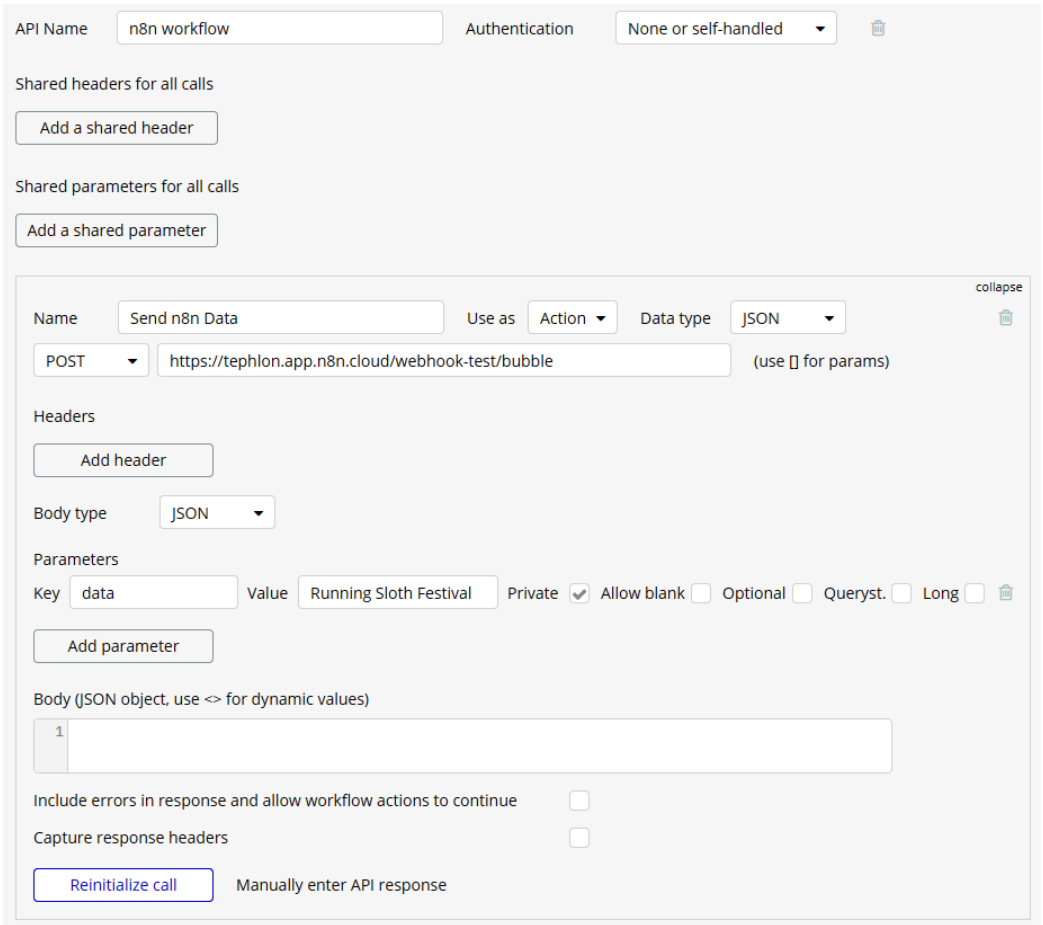

Figure 8.16 – Completed Bubble API Connector plugin

Now, whenever you are working on a workflow in Bubble, you can use the **n8n Workflow - Send n8n Data** action in the **Plugins** section, as illustrated in the following screenshot:

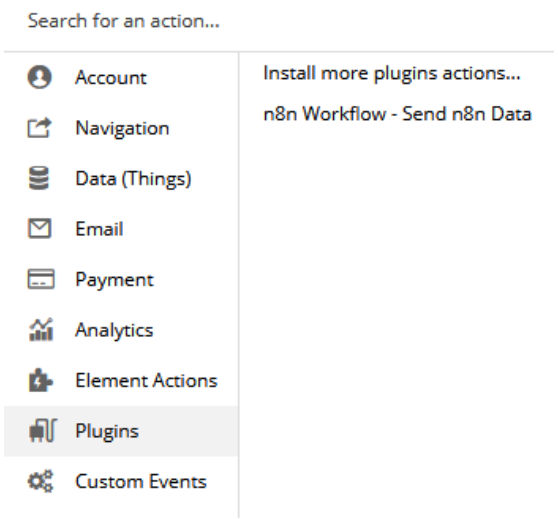

Figure 8.17 – New n8n action in the Bubble workflow editor

Now that you have configured both n8n and Bubble to work in harmony, you can extend Bubble however you wish with all the power of n8n, its nodes, and any of the services that n8n can access. This opens up your Bubble application to do almost anything that you can dream of.

Summary

In this chapter, we learned about the Bubble API. We worked with the Data API to manipulate information in Bubble and the Workflow API to execute workflows. We also learned how to do all of this using n8n, primarily with the **HTTP Request** node.

We also learned how to use Bubble to activate Webhooks in n8n and pass information between the two systems.

In the next chapter, using Bubble, we are going to be learning how to build the frontend of an application that will use n8n on the backend to process information.

Section 3 – Building the User Interface and Connecting the API

In this section, you will build the user interface for the application and bring all of the parts together into one complete application.

In this section, there are the following chapters:

- *Chapter 9, Building the User Interface of the Application*
- *Chapter 10, We've Only Just Begun*

9

Building the User Interface of the Application

While the main focus of this book is to teach you how to use n8n to build application workflows and connect various tools together, it is extremely important that your applications have a proper **user interface** (**UI**). This is the primary way that people interact with your product and is the biggest aspect of the **user experience** (**UX**).

Having spent time learning about how the Bubble **application programming interface** (**API**) works in the previous chapter, we have a good understanding of what happens with Bubble "under the hood." Now, we will be using Bubble to build a UI and a design model that will allow you to create your own UI.

This chapter covers the following topics:

- Implementing responsive design for your web app
- Working with events in Bubble
- Validating data in Bubble

- Designing the application structure
- Dealing with errors in Bubble

Once you have completed this chapter, you will know how to do the following:

- Design responsive applications using the Bubble **graphical UI** (**GUI**).
- Learn how the look and feel of an application can change the UX.
- Understand underlying data structures.
- Guide users in entering appropriate data to fit into data structures.
- Identify errors in applications and workflows.
- Proactively handle how errors are presented to users.
- Design a logging system to capture events and errors.
- Analyze data captured in logs for application improvement.

Let's start this chapter by looking at responsive design and implementing it into your Bubble web application.

Implementing responsive design for your web app

Responsive design is a methodology for building web-based applications that display information correctly, regardless of the device or screen/window size. The ability to view information on various devices and screen sizes has become increasingly important over the last decade. The following screenshot reflects this trend:

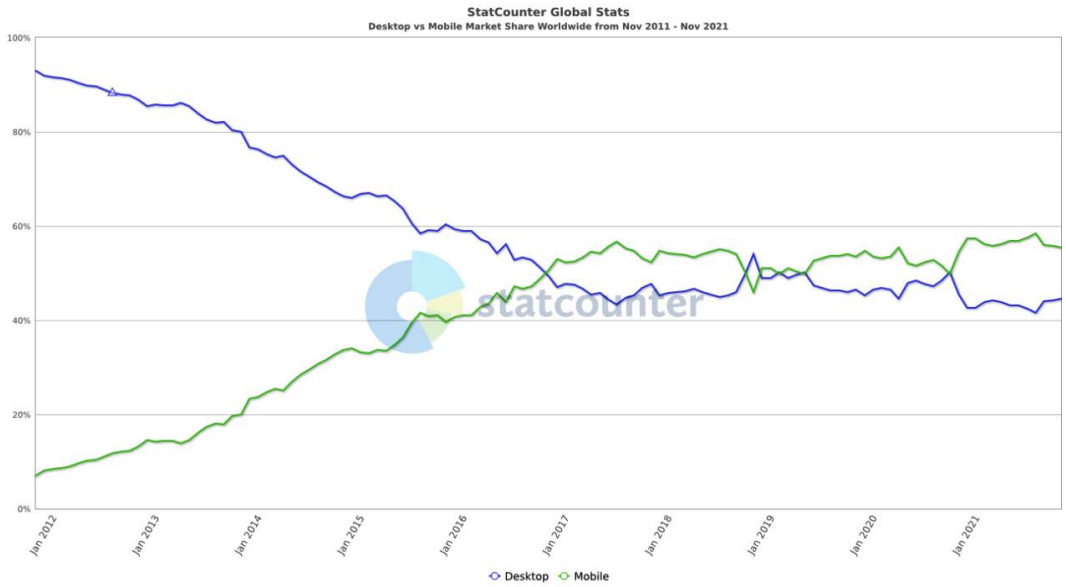

Figure 9.1 – Desktop versus mobile usage over the last decade

According to *StatCounter* (`https://gs.statcounter.com/platform-market-share/desktop-mobile/worldwide/#monthly-201111-202111`), 10 years ago, over 93% of people were accessing the internet using their computer. Today, that number has dropped to around 44%. More people have gone from large computer screens to smaller mobile devices to interact with web-based UIs. Because of this, it is critical that your web app looks good and is easy to use, regardless of whether you are using it on your 30-inch computer monitor or your 4-inch phone screen.

Responsive design factors

For your application to work responsively, here are some factors to consider when building your application.

Using the Responsive Viewer

In the **Design** tab of the Bubble application interface, you find the **Responsive** tab located to the right of the **UI Builder** tab. The **Responsive Viewer** allows you to see what your page looks like on different-sized devices.

Minimum width

Increase the minimum width of elements to use more of the page margins. Larger minimum widths make your design look better on smaller screens.

Fixed width

Some elements, such as icons, should have their width fixed. This prevents the elements from becoming too small or large as the screen size changes.

Maximum width

Buttons and inputs may look strange if they are too large. Correct this issue by setting the control maximum width side.

Margins

Consistent margin size is an integral part of the design aesthetic. Whenever possible, keep margin sizes consistent.

Collapsing margins

Remove the left and right margins around specific elements (for example, graphics; embedded elements) when the screen size gets smaller. This more efficiently uses valuable screen real estate.

Alignment

Align elements to either the left or right margin so that it "sticks" to one side. This makes your design more predictable as the screen size changes.

Hiding elements

When a parent element's (for example, a page) width drops below a specified size, do not show the element. With smaller screens, this provides a much better experience for the user.

Wrapping to the previous line

If a page size becomes large enough to accommodate the element on the previous line, it moves it to the previous line. This prevents large areas of white space along the right margin for larger screens.

Text element options

When working with text elements, there are a couple of other options that should be considered, as follows:

- **Cut off content if too tall**—Rather than making a text element taller as the screen shrinks, keep the text element height, hide unviewable text, and replace it with ....
- **Shrink if text gets shorter**—Reduces the height of a text element if there is white space.

Image/Google Map/shape element proportions

Use the **Keep element proportions** option to maintain the width and height ratio regardless of screen size.

Repeating group cell width

The **Current cell minimum width** option allows you to "stack" cells differently rather than just shrinking cells to a point where they become unreadable. You accomplish this by increasing the **Current cell minimum width** value.

Using the Responsive Viewer

By default, the Responsive Viewer is in large-screen mode, which shows you how your page will look with a screen that is 1,200 **pixels** (**px**) wide, as illustrated in the following screenshot:

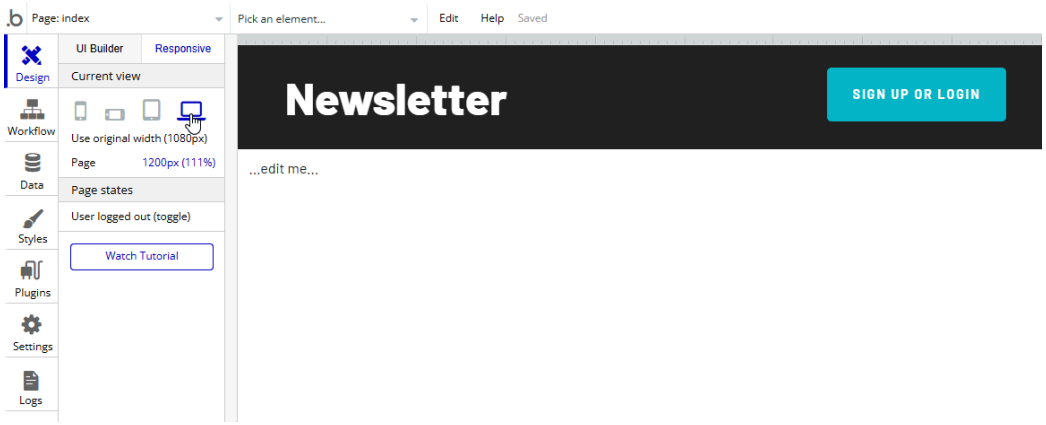

Figure 9.2 – Responsive Viewer in large-screen mode

You can change this view to see, for example, how the page will look when viewed from a mobile device such as a cell phone, as illustrated in the following screenshot:

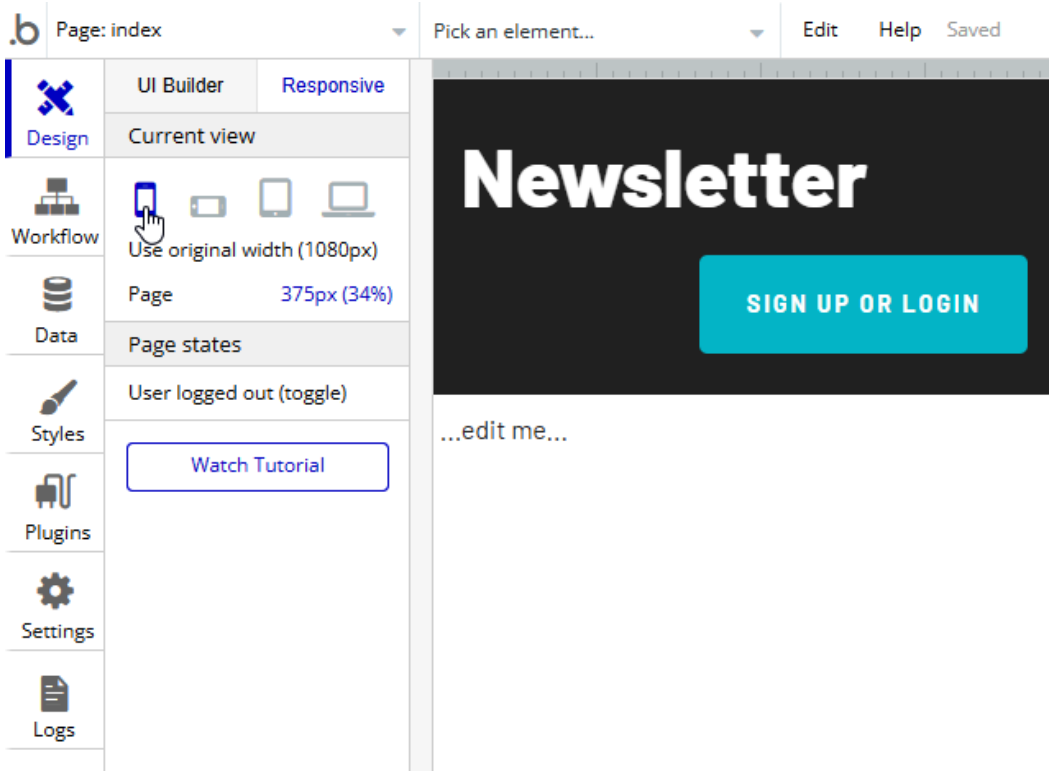

Figure 9.3 – Responsive Viewer viewed from a mobile

Throughout the design process when you are building your application, it is a good idea to come back to the Responsive Viewer frequently to ensure that your application continues to look good, regardless of the device that is being used.

Learning more

This has just been an overview of responsive design and there is a lot more you can learn about Bubble and how responsive design is implemented. If you wish to dig deeper into Bubble responsive design, see the *Building Responsive Pages* web page in the Bubble documentation (`https://manual.bubble.io/help-guides/building-a-user-interface/building-responsive-pages`).

Responsive pages are very important to the modern web application user and can make or break the usefulness of an application, but even the best responsive web application is completely useless if it is not designed to work with events properly. Fortunately, this is the next topic that we are covering!

Working with events in Bubble

An event is an action that occurs within the Bubble app to trigger workflows. Actions within the Bubble interface often trigger events, but systems can also trigger events outside of Bubble, such as n8n.

Event types

While there are a number of different types of events that exist in Bubble, most of them are designed to be used internally with Bubble and do not interact with systems outside of the Bubble environment. These internal events are important to understand when working with Bubble and we highly recommend that you familiarize yourself with them.

General events

These events are common to most areas of Bubble and are accessible accordingly. They are outlined here:

- **User Logged In**—This event fires whenever a user logs in to your Bubble application.
- **User Logged Out**—Actions associated with this event will be executed when the user logs out of the Bubble application.
- **Page Loaded**—A **page load** occurs when a person opens up a web page on their computer. This event is fired whenever this happens.
- **Do Every X Seconds**—It can be useful to perform an action repeatedly based on how much time has elapsed since the last time the event occurred. This event is filed every *X* seconds, where *X* represents the number of seconds between events.
- **When Condition True**—This event compares some parameter of the system (for example, day of the week) with a value (for example, Tuesday) and then performs an action only if the comparison is true (for example, the action will only be executed if the day of the week is Tuesday).

Element events

Element events are related to the actual UI itself and are designed to interact with the user via the web UI. These events generally respond to something the user has done while in the application. They are outlined here:

- **Element Clicked**—This event is fired when a user clicks on a specific element in the web UI.
- **Input Value Changed**—If a field in a form has a value that is then changed by the user, this event fires.
- **Map Marker Clicked**—Map elements in Bubble can have markers placed on them. This event fires when the user clicks on one of these elements.
- **Popup Opened**—Bubble enables you to display messages to the user in the form of popups. This event fires when one of these popups opens.
- **Popup Closed**—You can also fire an event when the user closes the popup that was displayed to the user.

Trigger events

Trigger events are special events in the Bubble system that occur when changes are made in the database. These events can reference data values before a change in the database occurred or they can reference the values after a change in the database.

A couple of things to remember about trigger events. First, these events run with full privileges and have access to all data in the system, not just the data of the user who triggered the event. Second, trigger events can only trigger a single action, and that action cannot be used to trigger other actions.

Setting up events

In Bubble, events are managed in the **Workflow** editor tab. You configure actions that occur when a specific event is fired from this page, as illustrated in the following screenshot:

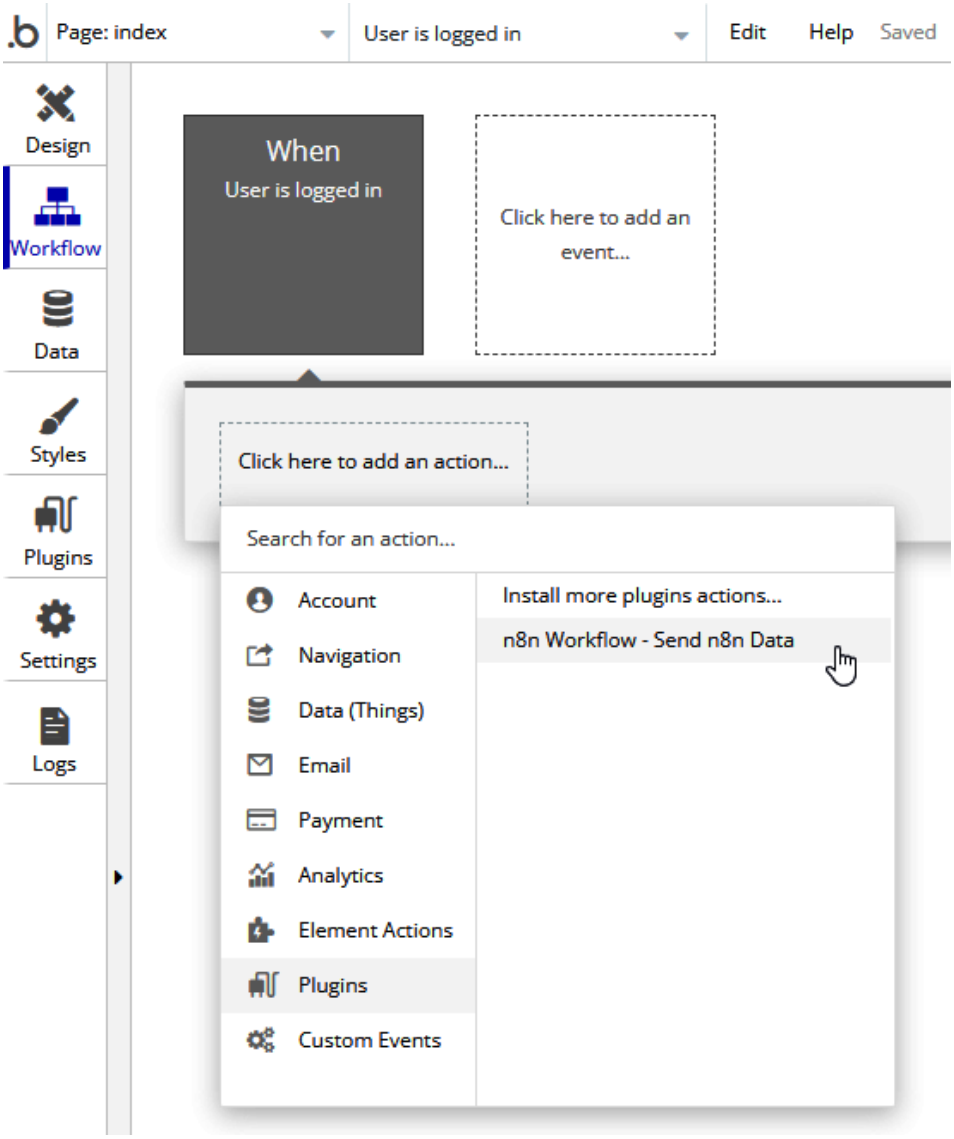

Figure 9.4 – Event in Workflow editor

For example, *Figure 9.4* is displaying how an individual can run the **n8n Workflows - Send n8n Data** plugin action when a user logs in to the Bubble app.

You can also run multiple actions when one event occurs, chaining the actions to execute one after the other. This allows you to perform complex actions while still keeping individual actions simple. This process is illustrated in the following screenshot:

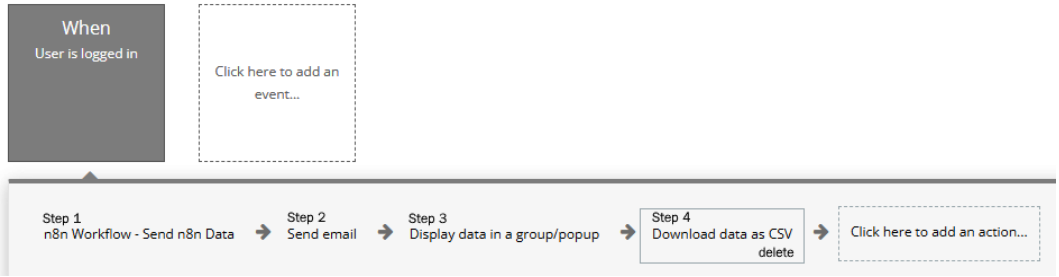

Figure 9.5 – Chaining actions to an event

Going deeper

This has just been a brief introduction to Bubble events, and you can dig a lot deeper into this topic. If you are looking to get a stronger understanding of events within the Bubble environment, I would suggest you start with *Building Workflows* (`https://manual.bubble.io/help-guides/building-workflows`) and *Events* (`https://manual.bubble.io/core-resources/events`) from the Bubble documentation.

Events are an important aspect of the Bubble architecture and are a critical part of proper data manipulation. But, if the data being entered into your Bubble app is faulty or the system confuses the different types of data, then suddenly the app becomes useless.

The next section helps to avoid some of these issues by showing us how to validate that the data being entered or analyzed is the correct data.

Validating data in Bubble

At the core of many applications is data. These applications rely on consistent, accurate, and structured data to provide analysis and insightful information. They also use the data to control different aspects of the application itself.

For these reasons, it is important that the data you are gathering and saving is as accurate as possible, especially when gathering that data from users.

Field types

The first way to control your data is through field types. Field types describe the data that can be entered into a specific field and restrict the information that can be entered into that field to that type of data. If you have a database background, these can be equated to data types within a database.

There are nine built-in field types in Bubble, as follows:

- **Text**—Any type of **American Standard Code for Information Interchange** (**ASCII**)-based text. Similar to a **variable character field** (**VARCHAR**).
- **Number**—Any numeric value, with or without a decimal. Similar to `FLOAT` or `DECIMAL`, but without the precision requirements.
- **Numeric Range**—Any numeric value between a lower and upper numeric value. This is similar to creating a database table and adding a `CHECK` range to an `INT` field value.
- **Date**—A calendar date and time. Similar to a database `DATETIME` data type.
- **Date Range**—Restrict the date value to a lower and upper date and time. Much like the **Numeric Range** field type, this is similar to creating a table with a `CHECK` range for a `DATETIME` data type column.
- **Yes/No**—A binary state that can be either **Yes** (`TRUE`) or **No** (`FALSE`). Similar to a `BOOLEAN` data type in a database. There is no `NULL` value for this field, as an empty field translates to a **No** (`FALSE`) value.
- **File**—A text value representing a **Uniform Resource Indicator** (**URI**) to a file. This is the equivalent of a database `VARCHAR` value that references the location of a file in a filesystem.
- **Image**—Similar to the **File** field type, this is a text value representing a URI to an image file. It also is the equivalent of a database `VARCHAR` value that references the location of a file in a filesystem.
- **Geographic address**—This is a unique field type that stores geographic information as text that has been validated by the Google Maps API. It appears as a street address, much like what you would find when mailing a letter.

Custom data types

You can also create custom data types. These are typically groups of field or other custom data types that work together. For example, a **meeting** type might be made up of a **date**, a **location**, and a **subject**.

Using the fields

Once you have field and data types defined, this can control the type of data that is required for data input. Depending on the type of data that is being stored in the database, you will have different options available to you for restricting the type of data that is acceptable to the field, as illustrated in the following screenshot:

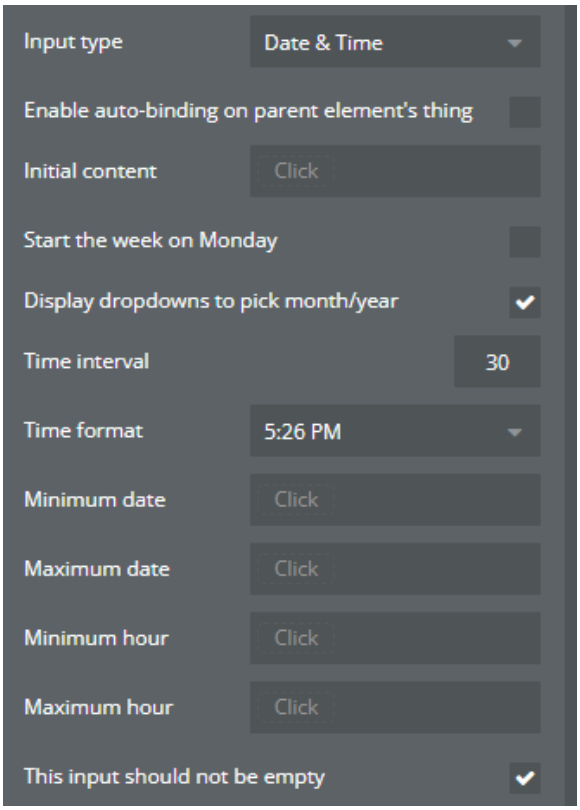

Figure 9.6 – Date & Time field restrictions

For example, *Figure 9.6* shows some of the restrictions that are available in the **Date & Time** input type. You can control the minimum and maximum values for the date and hour and require that the field not be empty.

More data validation

This is just the tip of the iceberg when it comes to data validation. There can be some very complex rules around the type of data that is permittable for an application, and it is worth your time to truly understand what these data rules should be and how to ensure your application is following those rules.

To get a deeper understanding of how to better manage your applications' data, I would suggest starting with *Working with Data* in the Bubble documentation (`https://manual.bubble.io/help-guides/working-with-data`) and use what you learn to maintain your data's integrity.

Now that we know we have the right types of data in the system, it is important that this data moves throughout the application properly and is stored predictably in the right places. Just like driving your car down the main hallway of your home and parking it in the living room, a poorly designed application structure can lead to unpredictable (and often disastrous) results. The next section covers how to plan and build the application structure.

Designing the application structure

An application does not just come out of thin air. A lot of planning and thinking goes into properly structuring an application so that it is both simple and useful. Some factors to consider when designing the application structure include the following:

- How many activities will the user need to perform?
- What type of data will be gathered?
- Does the user require authentication?
- Is the UX simple and logical?

It is extremely important that you think about how these (and other) factors affect your application and how your users interact with it. This helps to make your application easy to manage and maintain.

One of the first tasks that you should complete, even before you start coding a single page, is to create a flowchart that shows how a user moves through the application for each action that they will perform. This should show not only how the user interacts with the application but also how both the user and the application interact with the data in the application.

The following screenshot demonstrates such a workflow. This shows which data is being displayed/the user for each page, along with the path that the user must follow to get to each page:

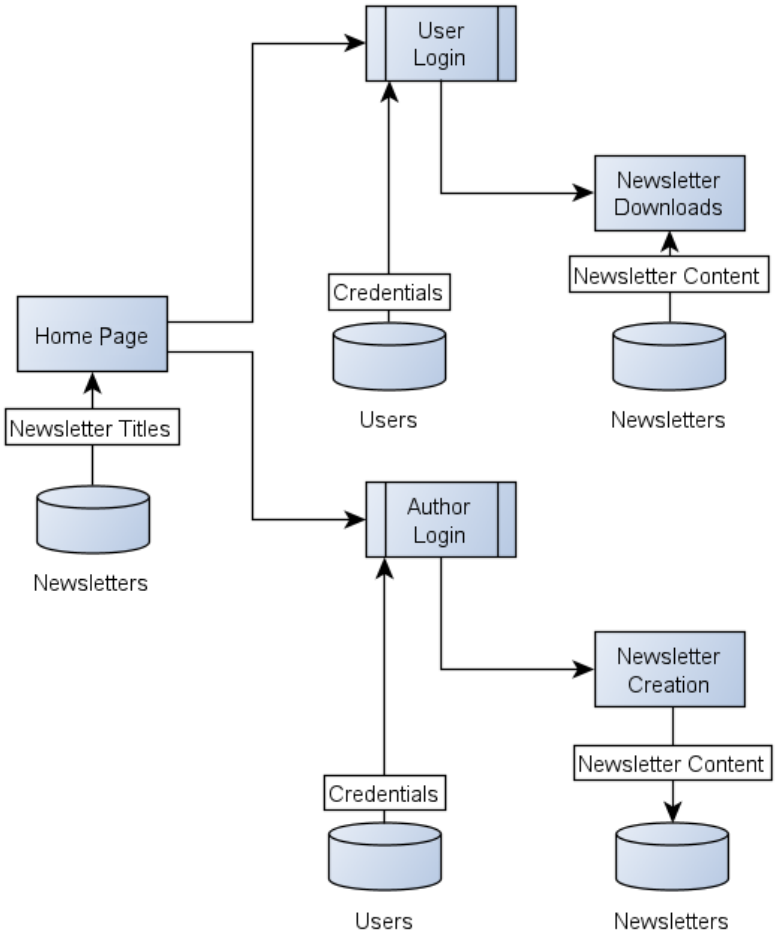

Figure 9.7 – Simple application workflow

This is an extremely valuable asset to have, especially if your team is large and different people are developing different parts of the application separately. It ensures that everyone understands how the application will work and how data will be managed.

It is also useful to ensure that the scope of the application does not change or expand. It makes it very clear what should and should not be designed.

Reviewing the design

Once you have completed the overall design, it is well worth having someone else on your team (or review it yourself if you are a team of one) to ensure that the design meets the goals and requirements of the application.

One excellent way is to create user stories that describe an activity from the perspective of the user. These stories are often in the form of *As a <user type>, I want to <goal to accomplish> so that <reason for activity>*.

For example, you may have the following user story:

As an author, I want to be able to edit newsletters that have already been published so that I can update information that may no longer be valid.

You can now take this user story and see whether there is a path in your workflow that will enable this to happen. If it turns out that there is no way for this to happen (which is the case for our diagram in *Figure 9.7*), you may need to go back and update your application design.

Once you are satisfied with the design of your application, ensure that it is marked as a final design and share it with your development team so that everyone is using the same design and aiming for the same goal.

But even with a really good design, things can still go wrong. This is where error handling and troubleshooting come into play.

Dealing with errors in Bubble

Applications are inherently complicated systems. There are events occurring behind the scenes, users behaving in unexpected ways, and unexpected platform upgrades that break previously usable code. Sometimes, it's a wonder that any applications work at all.

Because of this complexity, it is critical that applications be designed to anticipate ways that users can misuse the system while also dealing with errors that occur despite the best efforts of the application designers and developers.

Bubble has several tools available to help deal with errors that can crop up in the application, and several best practices will assist you if something goes wrong and you need to get things working correctly again.

Planning for user error

It is not uncommon for users to be a significant source of errors in your application. Either by simply not understanding what your application is attempting to accomplish or by maliciously attempting to bypass security and restrictions put in place, users can and will break your application. But you can minimize the impact that users have on your application by following a few best practices.

One of the best ways is to think like a user. Have someone test the application who is just a standard user and knows nothing about how your application works prior to rolling it out to the rest of the world. Carefully document how the user got themselves into the situation where they generated errors and how the error was handled. Then, update the workflow, deal with the error, and test again.

Locking down the application

There are a lot of different areas within the application that can cause problems if the user is allowed into them. For example, if the user can get into the user management portion of the system, they can add or delete users at random, causing all types of chaos.

Because of this, it is critical that the application be locked down to prevent random users from accessing parts of the application that they should not be able to access. This not only applies to the application itself but also to any management consoles, admin panels, and data management portals.

Detailed logging

If an error does occur, it is important that you have a way of determining what led up to the error. Building in a detailed logging service will allow you to trace what happened before the error occurred so that you can troubleshoot the problem and hopefully resolve what caused it.

Debugging tools

Bubble has two excellent tools that will help you to resolve problems that come up while developing your application.

The first tool is the **Issue Checker**. This appears as a red warning icon with red text, indicating the number of issues that exist in your application.

You can get more information about issues by clicking on the **Issue Checker**. This will pop up a list of all the issues. Clicking on any one of the items in the issue list will open up the screen where the issue exists.

The second debugging tool is the **Debugger**. The Debugger allows you to walk through your application step by step and see issues happening behind the scenes that the application user does not see.

To initiate the Debugger, simply open up any page of your application and then add `&debug_mode=true` at the end of the **Uniform Resource Locator** (**URL**). This will reopen your application page with the **Debugger** toolbar at the bottom, as illustrated in the following screenshot:

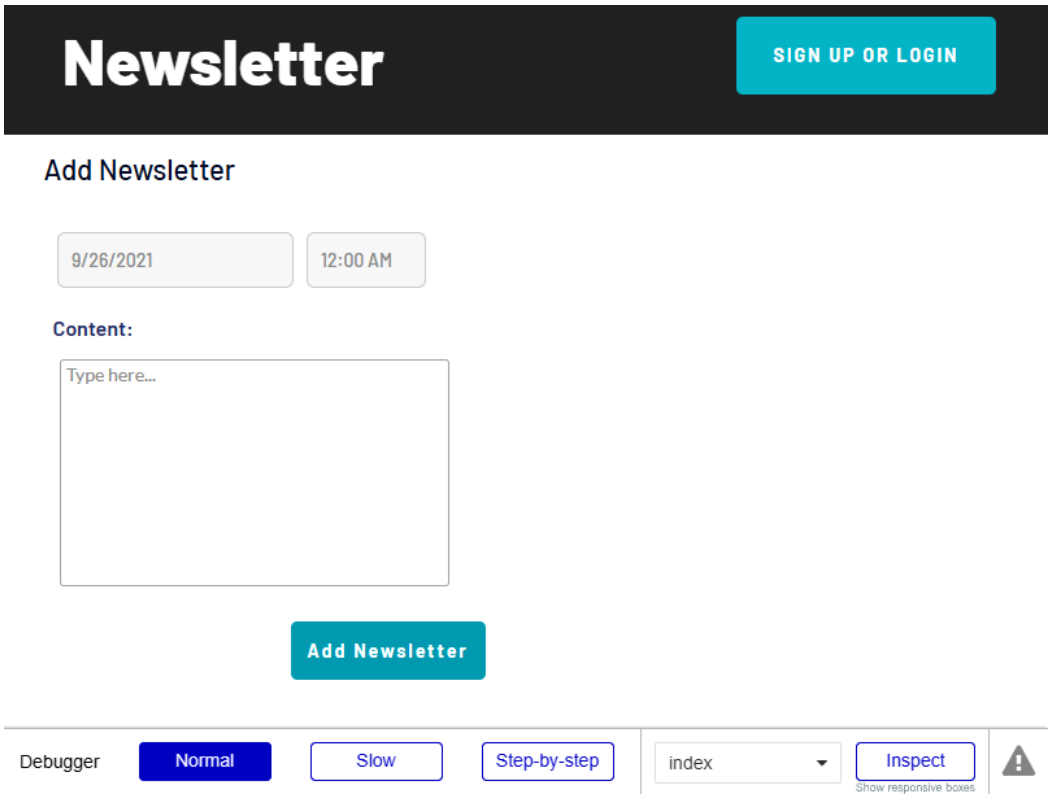

Figure 9.8 – Application with Debugger toolbar

This allows you to get a lot more detail about what the application is doing and provides you with clues for how issues may be corrected.

Summary

Designing and building a suitable web-based UI is one of the most critical aspects of your application build. It is the primary way that your users interact with you and your brand, and it is the ambassador for your organization. Thus, it is critical that it is as user-friendly, easily accessible, simple to understand, and error-free as possible. Hopefully, after having now completed this chapter, you are confident in the steps that you need to take in order to build this UI.

But what happens when you have your UI completed, all the workflows have been optimized, and the application still does not do what you need it to do? How do you extend the capabilities beyond what the Bubble development environment will allow you to do?

This is where n8n plays a major role in expanding these capabilities. We will be covering how to do exactly that in the next chapter.

10
We've Only Just Begun

It is sometimes hard to believe how much you've accomplished in the course of just reading a book. Sometimes, finishing a book can be both exciting because of the accomplishment as well as sad because the journey is over.

It's really exciting to see how far we have come in such a short period of time! Let's look at what we've learned from this book and help you find and start that next n8n project!

We've come a long way

Learning a new skill can be a daunting task. We look at where we are presently and calculate in our heads how far we have to go to reach our goal. It seems so far away. It seems almost too big to imagine what it will take to get to that finish line.

But sometimes, we spend too much time focusing on the finish line that we forget that it is the journey, not the finish line, that teaches us, molds us, and takes us to the next level. It is the journey that makes us *better, stronger, faster*.

This is why it is important for us to take a look back at what we have accomplished along the way and feel a sense of pride in how far we have come. Let's review what we have learned as we made our way through this book.

Introducing no-code tools

In section 1, we were introduced to a new way of developing code called no code. The concept behind no code is to move the building of applications away from software development teams and closer to the people who will actually use the applications, such as office workers or business analysts.

We also learned about the star of the show – n8n! We discovered how this automation tool can be used to connect different systems together, even if they were never designed to talk to each other. We learned how nodes in n8n are connected together in infinite combinations to manipulate data, connect systems, and create applications.

In the last chapter of this section, we built three different applications to demonstrate the power of n8n. These applications demonstrated the possibilities that are out there to connect and automate with n8n.

APIs and data

In the second section, we put a focus on APIs. An **API**, as you will remember, is an **application programming interface**, which allows different systems to communicate with each other without having to rely on people to perform manual tasks.

The first chapter taught us how to build our own API endpoints using n8n. This enables remote systems to activate n8n workflows on demand and allows n8n to create APIs for systems that may not have an API.

We then looked more closely at how n8n worked with data inside of its workflows. Data manipulation is a critical part of the flexibility inherent in n8n that makes it so powerful. The ability to transform data turns n8n into a "universal translator," allowing different systems to store information as needed.

Storing data can be as important as transforming data. So, we learned how to use no-code database solutions to store data for both short- and long-term use. These databases allow us to work with data at rest so that information is not lost between workflow executions.

Finally, we introduced how to use the Bubble no-code tool's API to access Bubble data and interact with workflows. This was the beginning of using Bubble as the web frontend for n8n.

Building the user interface

Up until now, the focus has been on using n8n behind the scenes to perform actions that most people do not see. In this final section, we focused on building out a way for users to interact with an application that uses n8n behind the scenes.

We showed you how to design an interface that will work well, regardless of the type of device it runs on. We also learned how to design the structure of an application and the data behind it to ensure that the application is easy to use and troubleshoot.

Finally, we connected the Bubble application to n8n in order to automate and process the information provided by the application.

Where to next?

Have you ever walked into a new restaurant for the first time and found it nearly impossible to decide what to order? There are so many combinations of entrées, sides, and desserts that it is impossible to land on one single selection to request from the waiter.

This is the problem that new n8n users are faced with. One of n8n's biggest assets, its flexibility, can also be one of the biggest roadblocks for new users who want to start creating with n8n. How do you choose?

Here are some tips for figuring out what you should do for your next n8n project.

Look for a problem to solve

n8n was designed to allow the user to solve everyday problems. It is very demoralizing when we build "a solution looking for a problem." So, look for those little daily annoyances that can be automated with n8n.

Does your hard drive keep filling up with old files? Build a workflow that examines all of the user files on your computer daily and then pushes any file that hasn't been modified in 2 years into cloud storage and removes it from your hard drive.

Do you keep forgetting to respond to emails in your inbox? Design an n8n app that will scan your email inbox and send you a message asking you what to do with the email and then deal with it appropriately.

Not sure what to make for supper each day? Have n8n pull a random recipe from your recipe list and create a shopping list for you.

Once you start looking at your everyday tasks as opportunities to make them more efficient or easier, you will soon find that there's no end to your ideas for n8n projects.

Dream big and start small

As we mentioned earlier, it is often the journey and not the destination that is important. The lessons that you learn along the way are often applicable to other challenges that you are experiencing.

So, think up some really big, wild project and just start! Break the project down into smaller subprojects and continue to break it down until it is something that you can do!

Here are some of the over-the-top ideas that I'm working on right now:

- Turning my truck into my own version of a cyber truck with n8n controlling out-of-this-world add-ons such as a rear projector and a pop-up video screen for instant drive-in movies, computer-controlled lasers, and individually addressed LED light upgrades
- Creating a bulk LEGO manager machine that will identify, sort, and catalog each part, determine what sets can be created from the LEGO in the inventory, and then put together all the parts for a set on demand along with a printout of the build instructions
- Designing a way to take a picture of books on a shelf, identify those books visually, then look up the value of each book second-hand, and upload the information to a website to put the books up for sale

Will I ever finish these projects? Who knows!

Will I learn some interesting skills and figure out how to use n8n in new and interesting ways? Absolutely!

Start an automation journal

Sometimes, ideas come to us at the strangest of times: in the shower, on a walk, when commuting to work, on a tour bus. It is important to get these ideas down as quickly as possible so that they don't get lost or forgotten.

But because the timing of these epiphanies is often not always conducive to standard good old-fashioned pen and paper, I purchased a waterproof notebook and keep it with me wherever I go. It is always within reach in the event that I come up with the next big idea!

Get ideas from others

Often, inspiration will come from what others have done (why do you think I listed my wild ideas earlier?). Take a look through some of the projects on the n8n websites or review some of the workflows created by other n8n users.

Starting your next project

Now that you have your new idea, how do you get started? This is usually the next roadblock that new n8n users run into. Getting past this can be difficult at it is easy to get overwhelmed by the potential immensity of the actual project (especially if it is a big, wild project).

In order to get you going, here are a few tips to get that new project launched!

Break it down

Sometimes, when looking at the whole project, it is just too much. How in the world can you expect someone to design that whole solution?

But if you break it down into smaller pieces then look at each piece, those pieces may be doable. If not, break each piece down even further until you recognize a piece that can be done!

Keep doing this until the entire project is complete!

Write it down

Trying to keep everything in your head for a project can be really difficult, especially if there are large gaps of time between when you can work on the project. By writing it down, you get the plans and ideas out of your head and down on paper.

Review n8n nodes

It can be really difficult to know what you can do if you don't know the tools that you have at your disposal. Since n8n upgrades frequently, it is good to upgrade your version of n8n frequently and look through not only the new nodes but also the changes that have been made to existing nodes.

Steal others' code

There is no sense in reinventing the wheel if someone has already built it. Take a look at workflows that others have already created and see whether you can use it or a part of it to complete a piece of your project.

Conclusion

We really hope that you have found reading this book as informative and educational as we did writing it. Feel free to reach out to us through the n8n community website if you have any questions or ideas for interesting projects.

Thanks for joining us on this journey!

Index

Symbols

A

B

C

D

E

F

G

H

I

J

M

N

O

P

R

S

T

U

V

W

Share Your Thoughts

Hi!

We're Jason and Tanay, authors of *Rapid Product Development with n8n*. We really hope you enjoyed reading this book and found it useful for increasing your productivity and efficiency in n8n.

It would really help us (and other potential readers!) if you could leave a review on Amazon sharing your thoughts on *Rapid Product Development with n8n*.

Go to the link below to leave your review:

`https://packt.link/r/1801817367`

Your review will help us to understand what's worked well in this book, and what could be improved upon for future editions, so it really is appreciated.

Best wishes,

Jason McFeetors

Tanay Pant