Learn IoT Programming using Node-RED

Begin to Code Full Stack IoT Apps and Edge Devices with Raspberry Pi, NodeJS, and Grafana

BERNARDO RONQUILLO JAPÓN

bpb

Learn IoT Programming using Node-RED

Begin to Code Full Stack IoT Apps and Edge Devices with Raspberry Pi, NodeJS, and Grafana

BERNARDO RONQUILLO JAPÓN

Learn IoT Programming Using Node-RED

Begin to Code Full Stack IoT Apps and Edge Devices with Raspberry Pi, NodeJS and Grafana

---

Bernardo Ronquillo Japón

www.bpbonline.com

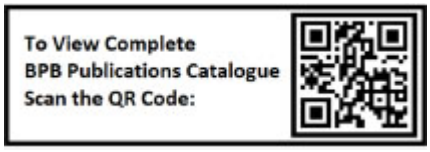
FIRST EDITION 2022

ISBN: 978-93-91392-383

LIMITS OF LIABILITY AND DISCLAIMER OF WARRANTY

The information contained in this book is true to correct and the best of author's and publisher's knowledge. The author has made every effort to ensure the accuracy of these publications, but publisher cannot be held responsible for any loss or damage arising from any information in this book.

All trademarks referred to in the book are acknowledged as properties of their respective owners but BPB Publications cannot guarantee the accuracy of this information.



www.bpbonline.com

Dedicated to

Lucas and Mario

My sons, who like me, are passionate for digital technologies

About the Author

Bernardo Ronquillo Japón is an Industrial Engineer with more than 25 years experience, expert in Artificial Intelligence, Robotics and IoT, writer of technical books as well as blogger of technology articles. Along his professional career he has been involved in a wide variety of sectors: Astrophysics, Industry, Aerospace, and currently Biotechnology. At present he works as Principal Software Architect in Universal Diagnostics, S.L.

As a result of his experience, Bernardo founded in 2011 The Robot Academy, where he developed Open Source hardware and software solutions for engineers and makers: Social Robot IO (2015) for the stimulation of children with Autistic Spectrum Disorders, Robot JUS (2016) in order to assist engineers to deepen ROS's technical insights using low complexity hardware, and IIoT All-in-One (2018) as the Industrial Internet of Things training package for helping companies in their digital transformation to Industry 4.0.

He is also a productive writer and likes to share his technical experience and thoughts in his blog **http://brjapon.medium.com**

About the Reviewer

Javid Ur Rahman is a distinguished database product manager and enterprise solution architect and has been actively involved in productizing and promoting cross-ecosystem collaboration in the Cloud Infrastructure, Edge and Analytics Platform space for over half a decade. He's focused on research and development of blockchain-based database algorithm designs and cloud-native-run engine development.

In his current role, he's taken the Enterprise Architect role in Fourth Square Inc, US Based Product, and Consulting Firm to new geographies.

LinkedIn **https://www.linkedin.com/in/jrahaman7/**

Acknowledgement

There are a few people I want to thank for the continued and ongoing support they have given me during the writing of this book. First and foremost, I would like to thank my partner Sofia for cheering me up while I was spending so many weekends and evenings on writing. I could have never completed this book without her support.

I would also like to mention my two sons, Mario and Lucas, who motivate me to write and transfer my passion for technology. I hope that my professional trajectory pushes them to keep on an always learning attitude. Thanks to this fact they may get highly qualified tech jobs that make them feel they are contributing to a better world.

Finally, I would like to thank BPB Publications for giving me this opportunity to write my first book for them.

Preface

There are many technical books dealing with the Internet of Things, albeit they put most of the effort on hardware selection and physical integration, with a pure coding approach to the software. This happens especially if the book is targeted to electro-mechanical engineers. But the point is that IoT is a transversal field covering many technological areas: electronics, information technologies, industrial facilities background, embedded software, wireless communication and software development; too many things for just one person, and too many topics to cover in a single publication. Hence this book put the focus on software development, applying the best practices in modern software engineering for building an IoT infrastructure.

So that readers' comprehension is complete, we also include the practical realization of the remaining core topics: hardware integration, embedded software, data processing, data streaming, storage in databases and end-users' visuals dashboard for information exploitation. How do we effectively transmit all that in a single book? By using plug-and-play components that we can quickly integrate in the infrastructure. In this way, we provide the key concepts that are essential to the global understanding, with the objective that reader can later apply them in more complex projects.

IoT hardware is addressed using plug-and-play components: Raspberry Pi as IoT device, and Sense Hat board as the collection of sensors integrated on a single board. So hardware assembly is as simple as plugging the Sense Hat on top through the 40-pins of the Raspberry Pi (GPIO, General Purpose Input-Output). GPIO are general in the sense that you can easily connect all kinds of boards: GSM internet connection, GPS board, air quality sensors hat, etc. In our case we will use a low cost board with plenty of sensors: temperature, pressure, humidity, and 3-axes gyroscope, magnetometer and accelerometer. The last 3 together are known as

Inertial Measurement Unit (IMU), and their measurements' combination provide the orientation and speed of a device in the 3D space. They are used in navigation systems to drive aircrafts, drones, satellites, spaceships, etc. For IoT infrastructures they are used to measure vibrations in machines, as well as orientation on mobile devices, being our smartphones an example of an ubiquitous extended application.

The primary goal of this book is to provide intermediate skills for IoT software development. Along the chapters, we will maintain an adequate equilibrium between hardware and software, so that the target audience-developers, engineers and makers/ hobbyists- may acquire a practical perspective on the execution of an IoT project. Over the 7 chapters, you will learn the following:

Chapter 1 describes the software architecture applicable to IoT projects, starting from a commonly used pattern in software engineering, .i.e. layered architecture. This is a key aspect that will prepare the reader to design and write efficient software.

Chapter 2 provides a practical introduction to NodeRED, the core visual programming tool for the project. There will be two NodeRED instances performing different functions: one in the IoT device, that will feed the streaming service with data coming from the sensors, and the other in the remote server, that will receive these data and apply transformations on them. Afterwards the results being written to a database.

Chapter 3 explains how to perform data acquisition from the sensors and transmit them via the streaming service. The software is written as a workflow of functional blocks on a canvas using NodeRED. The code to build will acquire data from simulated sensors, i.e. virtual Sense Hat.

Chapter 4 builds the data processing functionality. Several NodeRED flows will be developed to gather useful insights from the raw data supplied by the environmental sensors, as well as getting familiar with a simple approach to save time series in a physical file.

Chapter 5 covers the data storage functionality, as well as a dashboard visualization for the end-users. Data persistence function has not only the goal of providing safe storage, but also offers quick access to historical data for further analysis.

Chapter 6 finally introduces the actual IoT hardware-composed of the Raspberry Pi and the Sense Hat- and explains how to make them work together, proving it by writing several software snippets for testing.

Chapter 7 takes the modular software developed in chapters 3, 4 and 5 and plug them on the IoT hardware and remote server. All of them together will constitute the final application.

Code Bundle and Coloured Images

Please follow the link to download the
Code Bundle and the ***Coloured Images*** of the book:

https://rebrand.ly/fb99fd

The code bundle for the book is also hosted on GitHub at **https://github.com/bpbpublications/Learn-IoT-Programming-Using-Node-RED**. In case there's an update to the code, it will be updated on the existing GitHub repository.

We have code bundles from our rich catalogue of books and videos available at **https://github.com/bpbpublications**. Check them out!

Errata

We take immense pride in our work at BPB Publications and follow best practices to ensure the accuracy of our content to provide with an indulging reading experience to our subscribers. Our readers are our mirrors, and we use their inputs to reflect and improve upon human errors, if any, that may have occurred during the publishing processes involved. To let us maintain the quality and help us reach out to any readers who might be having difficulties due to any unforeseen errors, please write to us at :

errata@bpbonline.com

Your support, suggestions and feedbacks are highly appreciated by the BPB Publications' Family.

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 **business@bpbonline.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 **www.bpbonline.com**. We have worked with thousands of developers and tech professionals, just like you, to help them share their insights with the global tech community. You can make a general application, apply for a specific hot topic that we are recruiting an author for, or submit your own idea.

Reviews

Please leave a review. Once you have read and used this book, why not leave a review on the site that you purchased it from? Potential readers can then see and use your unbiased opinion to make purchase decisions. We at BPB can understand what you think about our products, and our authors can see your feedback on their book. Thank you!

For more information about BPB, please visit **www.bpbonline.com**.

Table of Contents

Index

Part - I

Creating IoT Infrastructures

CHAPTER 1

Introduction to IoT Applications and Their Software Architecture

Introduction

Many books refer to the trending technology, the **Internet of Things** (**IoT**) and the project-programming tasks as complementary activities, putting most of the effort in the hardware selection and physical integration. This is a classic perspective of an engineer that has to develop an IoT project.

But this is a transversal field, covering several areas: electronics, **information technologies** (**IT**), a background in industrial facilities/ workshops, embedded software, wireless communication, and software development; too many things for a single person.

Hence, the approach of this book is to provide primary/intermediate skills for the target audience: engineers as IoT builders—applying the typical project management skills in industrial projects, developers that need to integrate the application software with the embedded software in the IoT devices, and finally, makers (also known as hobbyists) that want to carry out a project on their own and obtain quick and tangible-looking results.

For this reason, we will provide a good equilibrium between hardware and software in the chapters. So that these technical profiles may acquire a practical and global perspective of what an IoT development means.

Taking the best practices from modern software engineering, you will learn how to build a robust, modular and functional application for an IoT project. After reading this chapter and covering the practical

exercises, you should be able to efficiently develop, test, integrate, and deliver software for other IoT projects.

You should acquire the expertise in small (incremental) steps to become an expert in the field. This requires time with many hours devoted to progressively complex projects.

For the sake of simplicity in the hardware part, the selected computer is *Raspberry Pi* **https://www.raspberrypi.org/**, and *Sense Hat* **https://www.raspberrypi.org/products/sense-hat/** the selected sensor bundle that includes both sensors and their conditioning electronics. The sensor board sits on top of *Raspberry Pi GPIO*, a collection of 40 *General Purpose Input/Output pins* that makes a straightforward connection with the external hardware in a plug and play philosophy from the electronics' perspective. These sensors provide the interface for a **single board computer** (**SBC**)-like the Raspberry Pi- with the physical world.

This approach simplifies the electronics part, and lets us devote most of our effort into the software: the one that is embedded in the SBC, as well as the code hosted in the external servers that processes data acquired from sensors.

From the software perspective, we should be aware that in recent years there has been a shift in the programming paradigm. Now, the task of creating a software is an integrated activity in the design workflow that engineers have to tackle. As a consequence, it is the same engineer who usually has to code the application. We should not expect him to have expert coding skills, but for sure he has the ability to write the logic of the application in schemes and drawings. Hence, what he needs are visual/modular tools that allow him to quickly develop a prototype and a **minimum viable product** (**MVP**) to show to the client.

There is a new generation of tools for visual programming that narrows the gap between the design language of engineers and the programming language of developers. This book is written around such kinds of tools.

For the purpose of carrying out the IoT project of the book, we will cover in depth the following ones:

- **NodeRED** (**https://nodered.org/**)**:** a visual programming environment conceived for programming the IoT.
- **Grafana** (**https://grafana.com/**)**:** a user-friendly web application, based on creating visual dashboards to expose graphs and tables that are built around the data coming from the IoT infrastructure. It also provides the functionality of creating alerts by defining rules based on the incoming data flow, hence, providing some degree of automatic operation of the IoT environment.

First, this chapter provides a short introduction to the IoT project that will be developed from chapter two.

Second, it describes the software architecture applicable to IoT projects, starting from a commonly used pattern in software engineering. This is a key aspect that will train the user to design and write efficient software.

Structure

In this chapter, we will cover the following topics:

- Introducing IoT applications
- The IoT project
- Layered architectures in software development
- Software architecture for IoT projects

Objectives

After studying this chapter, you will learn the essential concepts behind IoT projects and acquire a structured way of thinking that lets you connect functionalities with their concrete implementation as modular software. You will also be capable of defining the conceptual architecture for other IoT projects.

Introducing IoT applications

In this section, you will be introduced to the Internet of Things from the point of view of the developer. Following the approach of teaching by example, we will do this by briefly describing the project that will be carried out in the book.

The IoT project

Our project will consist of creating an infrastructure for *monitoring environmental conditions* in varied contexts: smart cities, industrial facilities, and even your home.

The following figure depicts the basic components that our IoT framework will have:

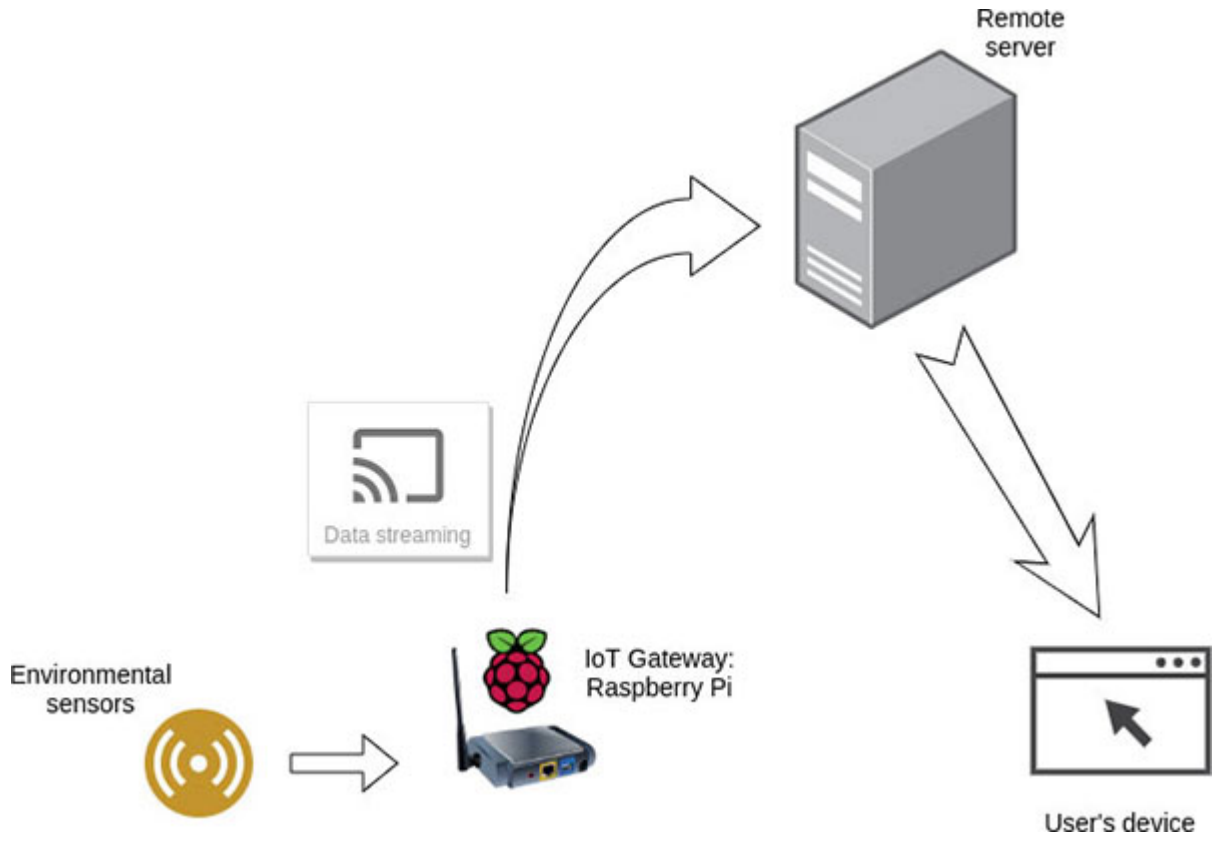


Figure 1.1: *Architecture of the IoT project*

Let's describe what each component is, and their purpose in this project:

- **Environmental sensors**: They are a set of commonly used sensors, such as temperature, humidity and pressure. All the sensors will be encapsulated in an electronic board called *Sense Hat,* which we briefly mentioned in the introduction of the chapter. It will be described in detail in *Chapter 4, Real time Data Processing with NodeRED*.
- **IoT device**: It's a non-standard computer that connects wirelessly to a network and transmits data over it. Due to this network interface, it also works as the IoT gateway because it will bundle all data feeds from sensors and will push them upstream for data processing in the remote server. The hardware of choice for the project is the single board computer Raspberry Pi that we will describe below.
- **Data streaming**: It is not a hardware component, but a software module that will provide the functionality of continuously streaming the sensors' data via an internet connection.
- **Remote server**: This is a powerful computer that will make data analytics. It will also prepare dashboards, with which the final user will monitor the environmental condition.
- **User's device**: This is the final point of the application. It can be any device that is able to run a common web browser: desktop PC, laptop, smartphone or tablet.

To finish this quick introduction, let's describe the central hardware of the project.

Raspberry Pi: The IoT device

The goal of such a device is to acquire data from external sensors. These may measure variables like temperature, humidity, pressure, etc. Data from sensors are acquired by the IoT device and streamed to the remote server.

Typical IoT devices are *minicomputers running Linux* operating systems. At the hardware level, they are able to interface with

physical devices as well as run a wide variety of applications, from software-oriented to sensing applications. Thanks to this, they provide high flexibility when designing the program that it will run. In contrast, common PCs or Linux servers are conceived to run any kind of software, and perform a wide variety of tasks.

The common approach in IoT projects is to use a **Single Board Computer** (**SBC**), where CPU, GPU, RAM memory, and physical storage are built on a single circuit board.

There are relevant advantages of this hardware design:

- The small size makes it easier to integrate any sensor that has to send data to an external computer (i.e., server in the cloud) for its operation and/ or monitoring of its condition.
- It is cheaper because of being built on a single circuit board, and the choice of electronic components that provide limited memory and computing capability. The software running in IoT devices is focused on data acquisition and actuators' operation, two tasks that do not require high performance characteristics.

Following this approach, the IoT device of choice for our project is the well-known board, *Raspberry Pi*, shown in the following *figure 1.2*. This is a computing platform designed for a specific, software-controlled task. It perfectly makes what we expect, because for an IoT application we need an embedded system that can run custom code for interacting with sensors and actuators.

Figure 1.2: *Raspberry Pi as the IoT device for the project (image courtesy: https://commons.wikimedia.org/wiki/File:Raspberry_Pi_4_Model_B_-_Side.jpg License CC-BY-SA 4.0)*

In this project, we will use Raspberry Pi because it is widely used in the industry and has a large community providing resources for solving common and complex issues. This aspect applies not only to the hardware, but also to the software, where most Raspberry Pi applications are built using open-source components that traditionally offer large community support.

An alternative hardware platform is a *microcontroller motherboard* like the well-known *Arduino*. It is simpler from the point of view of the software, because it does not require an operating system. One can simply flash the program that will carry the automated task (such as reading a sensor), and let it run in an infinite loop.

In the next section, we will cover the software pattern that will be used in our IoT application. You will see the importance of the design decisions for developing robust code that is easy to maintain. Thanks to its modularity, when you wish to extend your application, you will find that it will be a matter of adding new components in some of the layers.

Layered architectures in software development

One of the common architecture patterns in software engineering is the so-called *service layer's structure* or *layered architecture pattern*. The most distinct characteristic is that each layer is service oriented, i.e., there is separation of concerns. The application functionalities are described as services, where each service carries out a single function.

Each layer in the architecture provides an abstraction around the work that needs to be done on every particular business request. Each layer is composed of software components, each one being independent of the rest.

Layers also provide isolation of their specific functionality, i.e., changes made in one layer do not impact or affect components in other layers. Hence, a given layer does not need to know what other layers do to play its role within the application safely.

This feature allows building modular components that can be easily reused in other applications.

In summary, each layer has its specific role and responsibility. The interconnection between them makes the application work as a single unit, with which the user can interact via the command line or a visual interface (web application). In the following figure, a representation of the four standard layers is shown:1

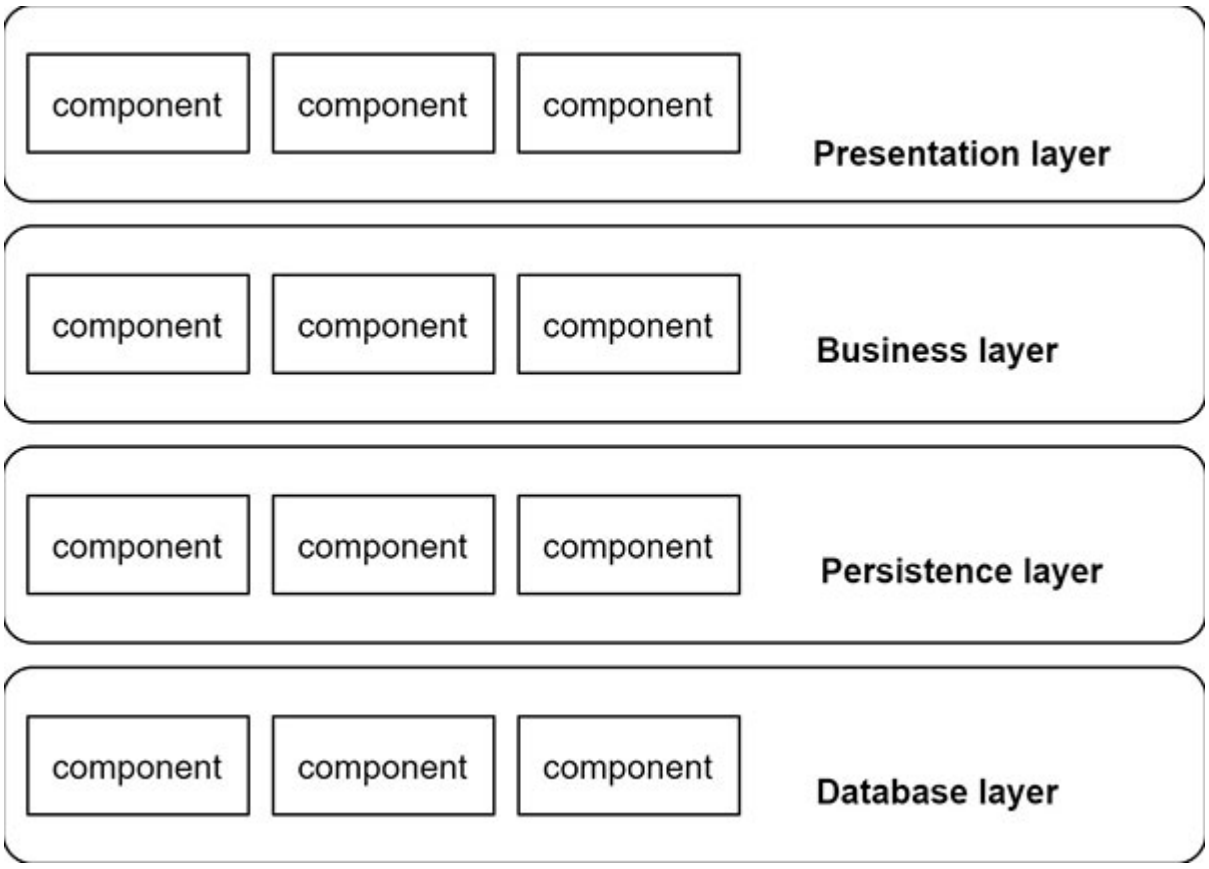


Figure 1.3: *Layered architecture of general-purpose applications*

Let's briefly describe what each layer is and what they provide to the application:

- **Presentation layer**: It stands for the visual interface that allows the user to interact with the application. The common implementation is a web app that only needs a browser to run, no matter which device is used (laptop PC, smartphone or tablet). Hence, the application is a cross platform, so there is no need for adaptation, according to the operating system running in the device.
- **Business layer**: This provides the software components that implement the business processes that the application is expected to deliver.
- **Persistence layer**: It builds and handles the requests that will hit the database for running CRUD operations of the business entities (temperature and humidity, for example). These

operations are a well-known standard that cover the operations of *Create, Read, Update* and *Delete* of entities. As you can easily infer, the first letter of each operation is concatenated in the acronym, making it identify all of them.

- **Database layer**: This layer receives the queries from the persistence layer and delivers the responses that the business entities will process afterwards.

In some cases, the business and persistence layers are combined into a single business layer, particularly when the persistence logic (i.e., SQL database queries) is embedded within the business components. We will apply this simplification to the pattern architecture for the IoT projects that we will present in the next section.

Note: At this point, do not worry if these descriptions are not completely understood. They still provide a high abstraction level. In the next section: Software architecture for IoT projects, we will recap on these concepts by applying them to IoT projects, so the explanations will be broken down with a practical focus.

These horizontal layers are built from components that consist of software modules. Each of them receives an input and provides an output to build the low-level functionalities of the application. The coordination of these low-level functionalities aggregates to produce the top-level application functionalities.

The separation of concerns occurs among the components in each layer. This allows us to easily build effective roles and responsibility models. The main consequences from the point of view of software development are the ease of development, testing, and maintenance of the application. In summary, this architecture pattern provides well-defined component interfaces and limited component scope.

Since the essential characteristic of the separation in layers is to assign specific roles, we can say that the design-driven concept behind this is the *single responsibility principle*.

So, let's proceed to the next section where all the concepts explained above will be detailed with respect to the practical components in an IoT application.

Software architecture for IoT projects

IoT applications are of a different nature with respect to the common web or mobile applications, because they ingest data from physical environments to build their functionality. Furthermore, the pattern architecture described in the section above is the same that applies to the non-IoT part of our application. This part is typically a web application whose software components are as follows (see *figure 1.4* below):

- **Backend infrastructure**: where data processing is performed to build the business functionality. This corresponds to the database and business layers of the pattern.
- **Front-end infrastructure**: where the visualizations that compose the user dashboard are created. This corresponds to the **Presentation Layer** of the **Application Architecture** in the figure.

Hence, *figure 1.4* shows the double-layered architecture proposed for IoT projects, where the IoT blocks at the bottom repeat the web application structure.

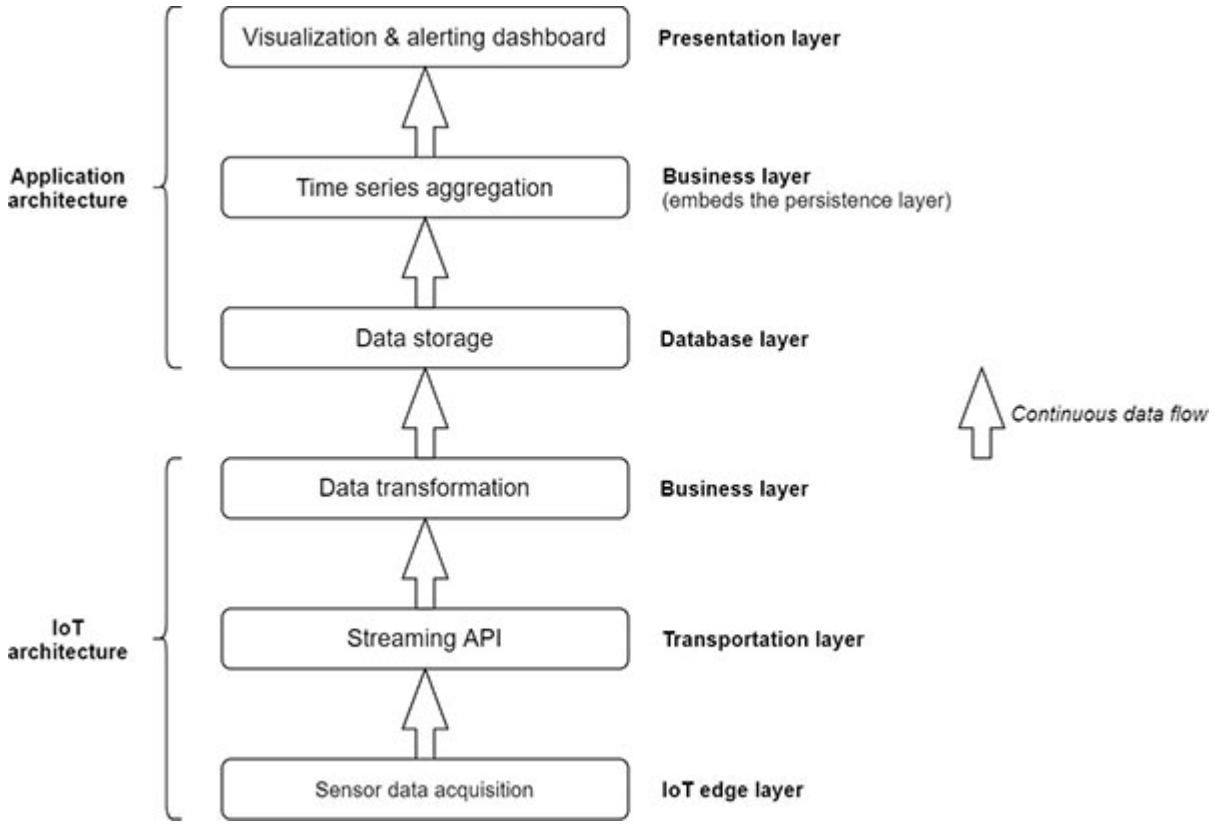


Figure 1.4: *Layered architecture for IoT applications*

So, let's describe what responsibility applies to each layer of the application:

- **Storage of the time series data**: This role corresponds to the database layer. It retrieves the stream of data, and provides the persistence functionality of the application.
- **Data analytics (like time series aggregation) to deliver processed data**: This responsibility corresponds to the business layer in the application architecture block. Be aware that this business layer is different from the business layer in the IoT block downstream, which we will explain in short.
- **Data analytics results visualization through dashboards**: This function corresponds to the presentation layer and occurs in both the backend server and the user's device. The backend builds the code to run on the client's device (laptop, smartphone, tablet, etc.), whose result is shown via a web browser.

Next, we will describe the IoT block, where one of its characteristics is how the data exchange is handled between layers. A permanent connection is established between IoT devices and remote servers, through which the sensors' data is acquired. That remote server is the one mentioned in the first section, *Introducing IoT applications*.

As it was pointed out in the previous section, each layer in the architecture has its own separate role/responsibility in the application, and the same applies to the IoT stack. At this level we can differentiate three responsibilities:

- **Data acquisition from the IoT devices**: At this level, we have a set of sensors whose measurements need to be acquired by the IoT devices. This function corresponds to the IoT edge layer in the IoT architecture block.
- **Streaming of data to remote servers**: Another device known as an IoT gateway will forward all the collected measurements through an internet connection, allowing their transmission to the servers. There is a component in the architecture called **streaming API**. It transmits the sensors' data to the remote servers. This role corresponds to the transportation layer in the IoT architecture block.
- **Data transformation:** This is to produce a curated stream on top of which the application functionality will be built. This responsibility corresponds to the business layer in the IoT architecture block.

Upward data streams feed the application components of the web app, distributed in the **Application architecture** block. Next, we will enter into the practical realization of this general architecture for our concrete IoT project.

Software architecture for the IoT application

Proceeding to the next level of detail, we will group the layers to provide a simpler and more compact structure.

First, we merge the two blocks, i.e., *Application architecture and IoT architecture*. This way, we will simplify and deal with a single stack. According to this, there are three groups of layers (refer to *figure 1.5* below for their visualization):

- **Application layer**: It will contain two layers of the application architecture stack:
    - The **Presentation Layer** will provide the visualization of the IoT dashboard in the web browser.
    - The **Business Layer** will send queries to the database (see the database layer below) to build the graphics and the summary tables that will be placed in the graphical interface.
- The **Middleware Layer** will contain a mix of layers from the application and the IoT architecture stacks:
    - The **Database Layer** is where the time series of sensors' data will be stored.
    - The **Business Layer** of the IoT architecture will process raw data provided by the sensors, and store the aggregated data (averages, standard deviations, etc.) as time series in the database.
- The **IoT Foundation Layer** will contain the last two layers:
    - The **Transportation Layer** is where the streaming API component will transmit data from the IoT devices to the business layer.
    - The **IoT Edge Layer** contains the embedded software running in the IoT devices.

In the following figure, you can see how each of these three layers are covered in terms of hardware and software components. It also provides the scope of every chapter in the book. The software modules are developed from *chapters 3* to *5*, according to what is shown in *figure 1.5*:

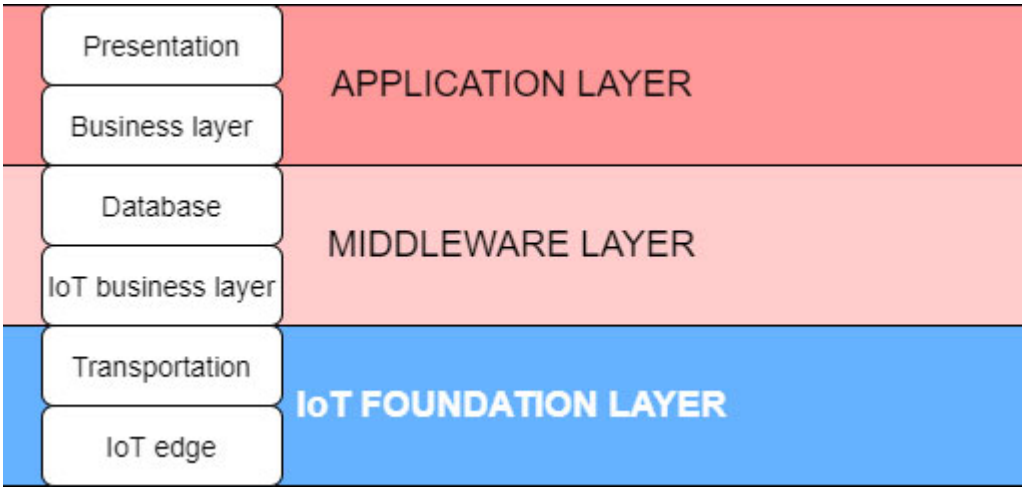


Figure 1.5: *Diagram for the layered architecture of the IoT application*

Having the above scope in mind, the chapters' outlines will be as follows:

- ***Chapter 2, Getting Started with NodeRED***, provides a practical introduction to NodeRED, the core visual development tool. There will be two NodeRED instances performing different functions:
    - In the IoT gateway, it will feed the streaming API with the data coming from the sensors.
    - In the remote server, it will receive data sensors and apply transformations on them that will be written to the database.
- ***Chapter 3, Data Acquisition and Real Time Streaming***, covers the IoT edge layer and the entry point to the streaming API (Transportation layer). So, it covers the whole IoT foundation layer. It explains how to perform data acquisition from the sensors and transmit them via the streaming API. The code that we will build in this chapter will run entirely in the IoT device.
- ***Chapter 4, Real Time Data Processing with NodeRED***, includes transformations from the raw data to the processed data. For learning purposes, we will apply a simple transformation consisting of calculating, at every timestamp,

the average temperature of the ten most recent measurements. Real-time data is retrieved at the destination point of the streaming API. This function is part of the middleware layer.

- ***Chapter 5, Storing and Graphing Data Streams with InfluxDB and Grafana,*** covers the application layer. This includes the database layer (uppermost of the middleware layer) and the presentation layer. It also includes the storage of the sensors' data in the database and the end-user interface based on visual dashboards.

Note: The practical implementation of *chapters 2* through *5* will be done in your laptop. We will provide the means to simulate the IoT hardware, including the sensors. The reason to do it this way is to skip the additional complexity due to the hardware, hence focusing on explaining how to build the software. Furthermore, this software will be the same that we will use in the IoT device and the remote server. That is the strong point of working with an IoT simulator that precisely reproduces the software interfaces with the IoT device.

- ***Chapter 6, The IoT Hardware Package,*** deals with the preparation of the IoT infrastructure. It explains how to set up the hardware (environmental sensors and the IoT device). Unit tests are carried out to check that they work properly, i.e., the sensors' data are acquired and available in the IoT device for processing.
- ***Chapter 7, The IoT Hardware Package,*** covers the practical implementation of the three layers described above: IoT foundation, middleware, and application layers. As a result, the IoT project will have been successfully executed, and you will have a nice application that provides temperature monitoring in real time.

Note: Implement row-level security that restricts the Salespersons to view the data only for their assigned regions.

So far, we have covered the conceptual part of the project. In the next chapter, we will put our hands into building the IoT application step by step.

Conclusion

In this introductory chapter of the book, we covered the basics of an IoT project. We described a general architecture pattern, commonly used in software development, and carried out the adaptation so that it fits as the IoT architecture pattern.

You have learned from a conceptual point of view how a software application is designed, being aware of how this pattern will be applied to our IoT project.

From *Chapter 4, Real time Data Processing with NodeRED*, onwards, we will dedicate specific sections to each part of the architecture matching the involved components of the IoT application.

Points to remember

- IoT devices act as an internet gateway that streams data from sensors to remote servers, where data analytics is performed.
- A layered architecture is a common pattern in software engineering in which each layer is service-oriented, having a specific role and responsibility from the point of view of the application.
- This kind of pattern can also be used in IoT projects by making the necessary adaptation to include the physical sensing functionality.

Multiple choice questions

1. **What is a Raspberry Pi?**
    a. an IoT device and nothing more
    b. a single board computer for running general purpose software and interacting with physical devices (sensors and actuators)
    c. a Linux server
    d. a desktop PC
2. **What is the scope of the Presentation Layer in the layered architecture pattern?**
    a. to visualize sensors' reading in the industrial facility
    b. to provide a web application that runs in any browser
    c. the visual interface for the user that allows him to interact with the application
    d. the layer in which the visual programming of the application is carried out
3. **For any application, what layer is mandatory?**
    a. the Business Layer
    b. the Presentation Layer
    c. the Transportation Layer
    d. the Persistence Layer
4. **What is the key factor that characterizes the Business Layer?**
    a. that it writes the application information to the database
    b. that it integrates with the Persistence Layer
    c. that it converts the user requirements into software code
    d. that it implements the high-level logic of the specific business whose processes it supports
5. **What do we understand by data streaming?**
    a. the continuous flow of the sensors' data to the IoT device

b. the continuous flow of the sensors' data between far points in the network
c. the continuous flow of the data analytic results to the user's device
d. the ability to transmit live events over internet

Answers

1. **b**
2. **c**
3. **a**
4. **d**
5. **b**

Questions

1. What are the advantages of a layered architecture pattern?
2. Why should you design your application with an architecture pattern in mind?
3. What is specific to IoT projects with respect to the layered architecture pattern?

Key terms

- **CRUD operations**: These are operations that apply to the business entities (temperature for example). These operations are Create, Read, Update, and Delete of those entities. As you can easily infer, the first letter of each operation is concatenated in the acronym, identifying all of them.
- **Edge IoT:** It refers to the processes that happen at the physical location where the IoT device acquires data from sensors. It physically occurs in places like factories or manufacturing plants. In the case of monitoring of environmental condition, sensors are located at the places

where temperature, humidity and/or air quality need to measured and controlled This is the case of smart cities, where a network of low-cost air quality monitoring sensors is deployed to monitor air quality and meteorological parameters across the city.

- **Embedded software**: It is a software that runs on the board of the IoT device. Its functions are initiated and controlled through machine interfaces.
- **IoT gateway**: It is a device at the edge where the sensors' data are acquired/aggregated and transmitted over the network for later processing in remote servers.
- **SQL**: It is the acronym of Structured Query Language, a standard language for storing, manipulating, and retrieving data in databases.
- **Streaming API**: The API is in charge of transferring data from one point to another in a continuous data flow. It sits on top of the software application that provides streaming functionality. The API is a specific interface so that developers can integrate that functionality into their applications.

CHAPTER 2

Getting Started with NodeRED

Introduction

This chapter provides a practical introduction to NodeRED, the core visual development tool in our project. There will be two NodeRED instances performing different functions: one in the IoT gateway, which will feed the streaming API with the data coming from the sensors, and the other in the remote server, that will receive data sensors and apply transformations on them that will be written to the database.

Following the steps to create a digital clock, we will introduce the NodeRED development tool to build IoT projects. You will first learn the basics behind the NodeRED architecture. Afterwards, a set of common practical operations will be covered: creating a flow, installing packages, and developing a visual dashboard for the *digital clock* project.

The concepts in this chapter are essential because NodeRED will power both the IoT application embedded in the *Raspberry Pi* and the backend server that will perform data processing.

Structure

In this chapter, we will discuss the following topics:

- NodeRED in brief
- Creating your first flow
- Finishing the digital clock flow
- Building the clock widget

Objectives

After studying this unit, you will have learned to install and set up NodeRED in a Linux PC, create simple programs in NodeRED—called

flows, and present the results in a dashboard **User Interface** (**UI**).

Technical requirements

The code for this chapter is hosted in a GitHub repository which you can find at **https://github.com/Hands-on-IoT-Programming/chapter2**

In the following subsections, we will cover the technical requirements that you have to comply with your laptop to make NodeRED run.

Operating system

Although you could install the application in any Linux distribution, we encourage you to use Ubuntu 18.04, the **Long Term Support** (**LTS**) version whose manufacturer, Canonical, will support up to 2023. All the examples in the book have been tested with it. The basic system requirements are as follows:

- Python >= 3.5
- Node >= 8

Python programming language ships by default with *Ubuntu*, while the *Node* runtime environment needs to be installed, and it will be covered below. First, we need to install *Git* in order to clone the code samples of the book.

Cloning the code

The *Git* tool is the most used in the developer community for the control version of software. We will use it to get the latest version of the code.

To get the code for this chapter, follow the next steps:

1. Update the software sources and install Git by typing in a terminal the following two commands:

```
$ sudo apt update
$ sudo apt install git
```

2. After finishing the process, go to your home location and create the folder where you will host the code of every chapter:

```
$ mkdir -p ~/book_hands-on-iot
$ cd ~/book_hands-on-iot
```

   The symbol ~ refers to your home directory, i.e. `/home/ubuntu`, if your username is `ubuntu`. It is a common shortcut whenever you want to make an operation involving your home path.

3. Clone the code using the installed Git tool:

```
$ git clone https://github.com/Hands-on-IoT-
Programming/chapter2.git
```

4. Then, change to the folder of the code samples for this chapter:

```
$ cd chapter2
```

5. Finally list all the files in such a location to check that they are present:

```
$ ls -la
```

   The output should be similar to this:

```
total 28
drwxrwxr-x 4 ubuntu ubuntu 4096 abr 5 13:34 .
drwxrwxr-x 10 ubuntu ubuntu 4096 abr 5 13:46 ..
drwxrwxr-x 5 ubuntu ubuntu 4096 abr 5 13:35 app
drwxrwxr-x 8 ubuntu ubuntu 4096 abr 9 11:27 .git
-rw-rw-r-- 1 ubuntu ubuntu 1646 abr 5 13:34 .gitignore
-rw-rw-r-- 1 ubuntu ubuntu 1081 abr 4 09:13 LICENSE
-rw-rw-r-- 1 ubuntu ubuntu 68 abr 4 09:13 README.md
```

The last point to get your system ready to run the examples is to install the runtime environment, **Node**, under which NodeRED will run.

Node

Node is an open source server environment-based on *Chrome v8* engine that will allow you to run the *JavaScript* code. It is commonly used to run backend applications, making life easier for developers,

since the frontend code running in the web browser is also commonly written in JavaScript. In this way, you can code both the frontend and backend of an application with the same programming language.

IBM recommends users to run NodeRED with Node version 10.x, although prior **Long Term Support** (**LTS**) version 8.x is still supported. To avoid any compatibility issue, we will use 10.x as suggested in **https://nodered.org/docs/faq/node-versions**. If for any reason you need a more recent version, you may run under Node 12.x, which is also supported by NodeRED. In summary, both 10.x and 12.x are the currently supported Node versions, 10.x being the recommended one.

Although we could download Node 10.x from the official website, we will use the package n instead, which allows us to easily switch between node versions.

1. Install the Node package manager, the key tool to manage the dependencies of your Node projects:

   ```
   $ sudo apt install npm
   ```

   This will also install a legacy version of Node in the system folder `/usr/bin`. We will override it to a more recent one by installing the versatile n package, as was mentioned above.

2. Since n is a Node package, install it globally using `npm`:

   ```
   $ sudo npm install -g n
   ```

   The output in the terminal will tell us the location of the binary as well as the installed version of Node:

   ```
   /usr/local/bin/n -> /usr/local/lib/node_modules/n/bin/n
   + n@8.0.1
   ```

3. Install *Node 10.x* using n:

   ```
   $ sudo n 10
    installing: node-v10.19.0
     mkdir: /usr/local/n/versions/node/10.19.0
     fetch: https://nodejs.org/dist/v10.19.0/node-v10.19.0-
     linux-x64.tar.xz
   ```

```
installed: v10.19.0 (with npm 6.13.4)
```

4. Check the installed version:

```
$ node -v
```

5. If you wish to install another version, simply run:

```
$ sudo n 12
```

6. The new active version will be 12.x. Check it again with the node version command above, i.e. `node -v`
7. To switch back to 10.x, simply repeat the activation command (this time it will not need to download the binary, since it is already present in your system):

```
$ sudo n 10
```

At this point you have your Ubuntu OS with Node. Then you are ready to install and execute NodeRED.

NodeRED in brief

The official documentation at **https://nodered.org/docs** is by far the best resource to learn NodeRED. In addition, we can search the NodeRED library for all of the available packages at **https://flows.nodered.org**.

Whenever we install a contributed package, we will provide the specific URL of the NodeRED library to have a quick understanding about what it does and how to use it.

Installation in the host OS

Follow the next steps to install NodeRED in your laptop:

1. Change to the `app` folder of the sample code, and install the dependencies:

```
$ cd ~/book_hands-on-iot/chapter2/app
```

   There, you will find a file called `package.json` that contains the dependencies of the project. The following snippet shows that part

of the file:

```
"dependencies": {
 "node-red": "~1.0.4",
 "node-red-admin": "~0.1.5",
 "node-red-dashboard": "~2.19.4”,
 "node-red-contrib-device-stats": "~1.1.2"
}
```

Let's briefly explain what each package provides (we leave for the next chapter the explanation about the version numbering):

- **`node-red`** is the main component, and has the code you need to run the environment.
- **`node-red-admin`** is a command line tool to administer the NodeRED application. It will allow, for example, to set a password for securely accessing the user interface in the browser. You can find details about the package at its **node package manager** (**npm**) page **https://www.npmjs.com/package/node-red-admin**
- **`node-red-dashboard`** is an essential contributed package, since it gives us the tools to build interactive dashboards and draw graphics. It is strongly recommended that you go through its npm page, at **https://www.npmjs.com/package/node-red-dashboard,** for a basic understanding of the functionalities it provides. There is the equivalent page also in the NodeRED library site, i.e., **https://flows.nodered.org/node/node-red-dashboard.**
- **`node-red-contrib-device-stats`** is a sample package that provides the consumption of resources of the system where NodeRED runs (CPU and RAM). We will use it as the Hello World example for getting started. Its npm page, located at **https://www.npmjs.com/package/node-red-contrib-device-stats**, contains the relevant information about it.

2. To install these dependencies, you only have to run from the **`package.json`** location:

```
$ npm install
```

3. Generate the hash for a password of your choice:

```
$ node_modules/node-red-admin/node-red-admin.js hash-pw
Password: <type the password>
$2b$08$a3BqIzZBlX1ZZGymCPCeV.JNy0nEmyOIRdiN.hDjxXaHMzPk1Z/uO
```

The output is the hash of the typed string (Raspberry in the example) to include in the **`settings.js`** file, together with the username you wish to have, i.e., admin for the example:

```
adminAuth: {
 type: "credentials",
 users: [{
  username: "admin",
  password:
  "$2b$08$a3BqIzZBlX1ZZGymCPCeV.JNy0nEmyOIRdiN.hDjxXaHMzPk1Z
  /uO",
  permissions: "*"
 }],
 default: {
  permissions: "read"
 }
},
```

Note: The file settings.js resides in the same location as package.json and it defines all the options for your instance of NodeRED. Within the file, all of them are commented out so that you can easily adapt settings to your needs

4. Finally launch the NodeRED application from the **`app`** folder:

```
$ npm start
```

Tip: The default directory for the user data is located at ~/.node-red. Otherwise, you should specify the desired location in the `settings.js` file with the option `userDir: '<PATH>'`. In the code of this chapter, we have set this option so that `PATH` is a

folder called data under the app folder, i.e. `~/book_hands-on-iot/chapter2/app/data`

5. Wait several seconds until the launch process finishes, something that you can easily track by following the messages in the terminal. Then, visit the URL **`http://localhost:1880`** and you will see how the NodeRED flow editor appears.
6. Log into the app (by clicking on the top-right user icon) with user admin and the password you set, i.e., raspberry. Afterwards, you will see the sample preloaded flow, shown in the image below, that collects every second the system resources usage, reporting the CPU load and memory. To see this feed of data, click on the icon identified as debug in the image:

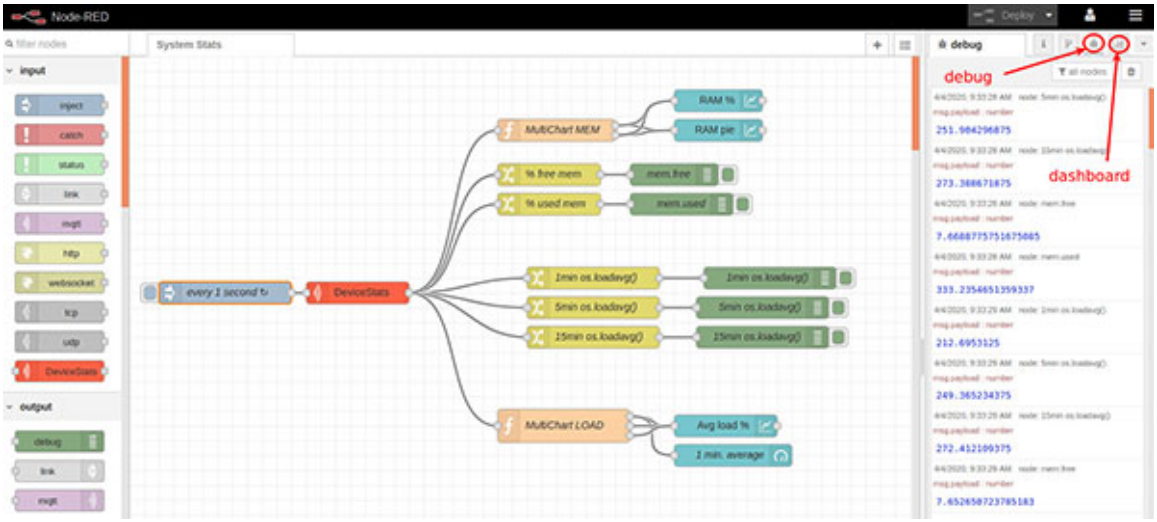


Figure 2.1: *Sample NodeRED flow*

7. Next, click on the icon identified as dashboard (see the red circle to which the dashboard arrow is pointing in the previous image). You will see how the tab in the right column of the screen changes to the **`dashboard`** pane with the layout tab in the front, as shown in the next screenshot:

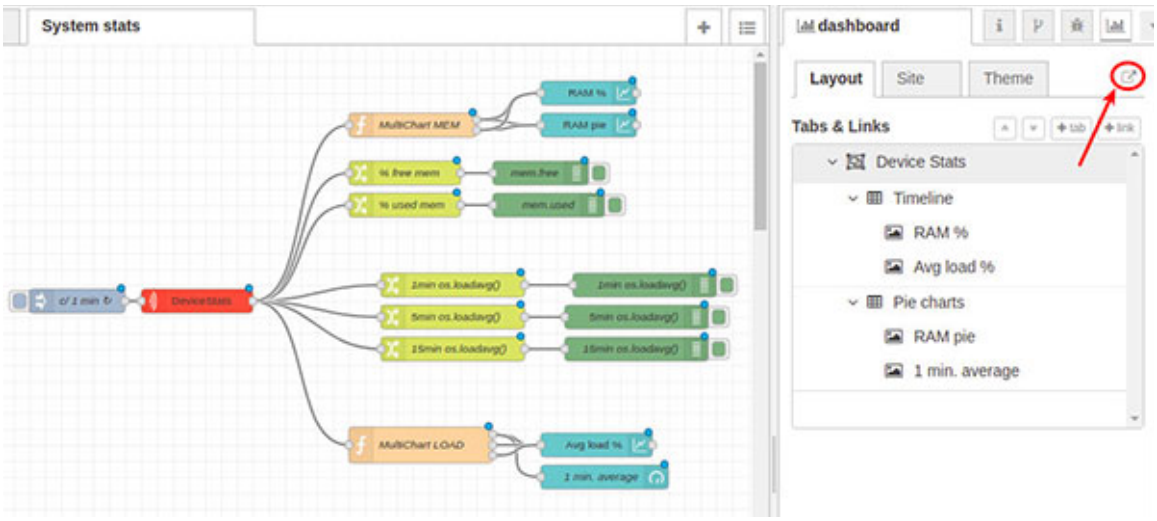


Figure 2.2: *Layout tab of the dashboard pane*

8. Then, click on the top right link icon (again a red circle to which the arrow is pointing) and a new browser window will open showing two graphs, one pie chart and one gauge like in the following screenshot:

Figure 2.3: *Sample dashboard user interface (UI)*

Tip: You can also access the dashboard UI by writing in another browser tab the URL **http://localhost:1880/ui**.

In the practical example of this chapter, you will learn to build a similar dashboard, but instead of system resources usage you will acquire environmental data—temperature, humidity and pressure—and will plot the time series using similar graphs.

NodeRED architecture in brief

We assume you have no previous knowledge of NodeRED, and to be as didactic as possible, we will describe its architecture with an intuitive analogy. NodeRED flows can be visualized as a system of pipes through which the water flows. Take the visual example in the next picture:

Figure 2.4: *NodeRED flows analogy (image courtesy:* ***https://pxhere.com/en/photo/1135421*** *License CC0 Creative Commons)*

The pipes are the wires connecting nodes in NodeRED. Like water or other fluids flowing through the pipes, JSON formatted messages are the particles that circulate inside the NodeRED wires. Just as you can

measure the flow inside a pipe in liters per second, for NodeRED the equivalent is the number of messages per second that circulates through a wire: the more liters per second through a pipe, the more messages per second through a wire.

At this point, you may guess that the fluid of our system is the data, packaged as JSON messages. And the data really flows, as it sounds!

The other core concept to explain is the node, the point in which one or more wires coincide in a NodeRED flow. In systems of pipes,/ the analogy can be established with the pipe joints, where two or more of them intersect as shown in the following picture:

Figure 2.5: *Sample NodeRED flow (image courtesy:* ***https://www.needpix.com/photo/1333874/pipes-joints-bolts-thread-plumbers-business-job-profession-repairman*** *License CC0 Creative Commons)*

For example, in the case of two *input wires* carrying messages, you can combine them to supply a larger message that is fed through the *output wire*. It is like adding two flows in a system of pipes, where the output flow is the sum of the incoming two.

In the case of NodeRED, one input wire could be supplying messages with temperature and timestamp, and the other with humidity and another timestamp. To combine them means to build a new message called environment that contains both the temperature and humidity,

and their respective timestamps. The flow of output messages would consist of the timeseries of environmental conditions.

This combination is what performs the JavaScript code wrapped by the node, i.e., you insert the code snippets to transform the data that arrives into another kind of data. NodeRED is user-friendly and instead of having to write the JavaScript code, it provides a collection of nodes that you program through forms, avoiding to deal directly with the code.

With these concepts in mind, you are going to find now how easy it is to start working with NodeRED. So, let's describe its user interface.

NodeRED web interface

After the quick launch of the app we made in the first subsection *installation* in the host OS, it's time to get familiar with the **User Interface** (**UI**) editor. It is divided into three regions, as shown in the following screenshot:

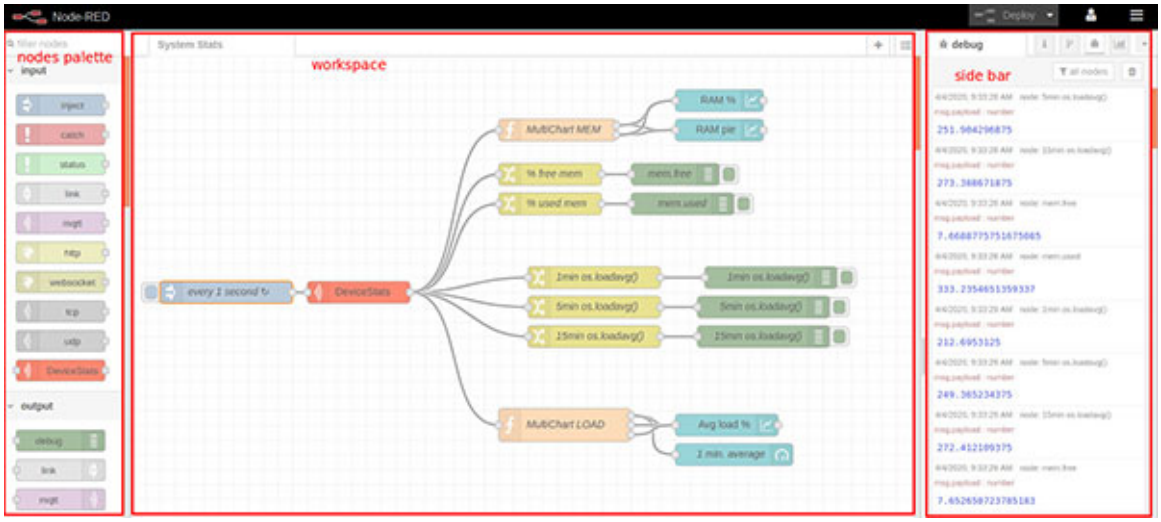


Figure 2.6: *NodeRED web-based editor*

Let's describe each one of them:

- **Workspace region**: This is the largest area and it will be the place where you will design your workflow, i.e., nodes connected by wires. Following the analogy presented in the previous subsection, you build a flow in NodeRED in a similar manner as you assemble a set of pipes.

- **Nodes palette**: It is located in the left part, and from here you can select the node you need to perform a certain operation, like combining data of pressure and temperature in a single message. Nodes are grouped in useful categories—*input*, *output*, *function*, *storage*, etc.—and you will have the opportunity to get familiar with them by following the many examples that we will explain in the book.
- **Side bar**: Located in the right part, there is a collection of tabs that you can select to be at the front as you need by clicking. Each tab is a useful tool that NodeRED provides. Let's briefly describe them:
    - **Node information**: It gives you the practical information of the node you select in the workspace: what it does, inputs, outputs, and what are the parameters that you can fill or modify in its form. When you double-click on a node, a window for the form is open so that you can program the behavior of such node.
    - **Project history**: This tool is used when you need to control the changes and versions of the flows under a NodeRED project. It runs on top of the Git control version tool. In the next chapter, we will explain what NodeRED projects consist of and how we can manage them.
    - **Debug messages**: This tool will be the one that you will use the most. It will show you, in real time, the messages that are flowing through the wires. Hence, it is an essential tool to debug your flow as you build it in the workspace.
    - **Dashboard**: This tab is added by one of the dependency `node-red-dashboard` that we installed with NodeRED. This package allows you to easily create interactive dashboards with live data. Just as the `debug` messages tab will be the developer's tool, the `dashboard` tab will be the UI designer's tool. Hopefully both roles will be played by you, because the NodeRED approach lets you cover both the phases of the software development process without being an expert.

These tabs `debug` messages and `dashboard` and let you guess that NodeRED has been conceived as a prototyping tool to quickly build an app. This is the main reason why so many people are contributing NodeRED modules, adding new node types to the Nodes palette, and giving you the opportunity to create an IoT application without even writing a single line of code.

Tip: The combination of developer and UI designer roles is what makes NodeRED suited to be a prototyping tool, letting you quickly sketch your ideas into a working app to show to your customers and colleagues.

After the description of the NodeRED interface, let's launch the application in the same manner that we did in the previous section:

```
$ cd ~/book_hands-on-iot/chapter2/app
$ npm start
```

As shown in the next image, you will be prompted to create a project now. Skip this option now, since we will cover it in the next chapter.

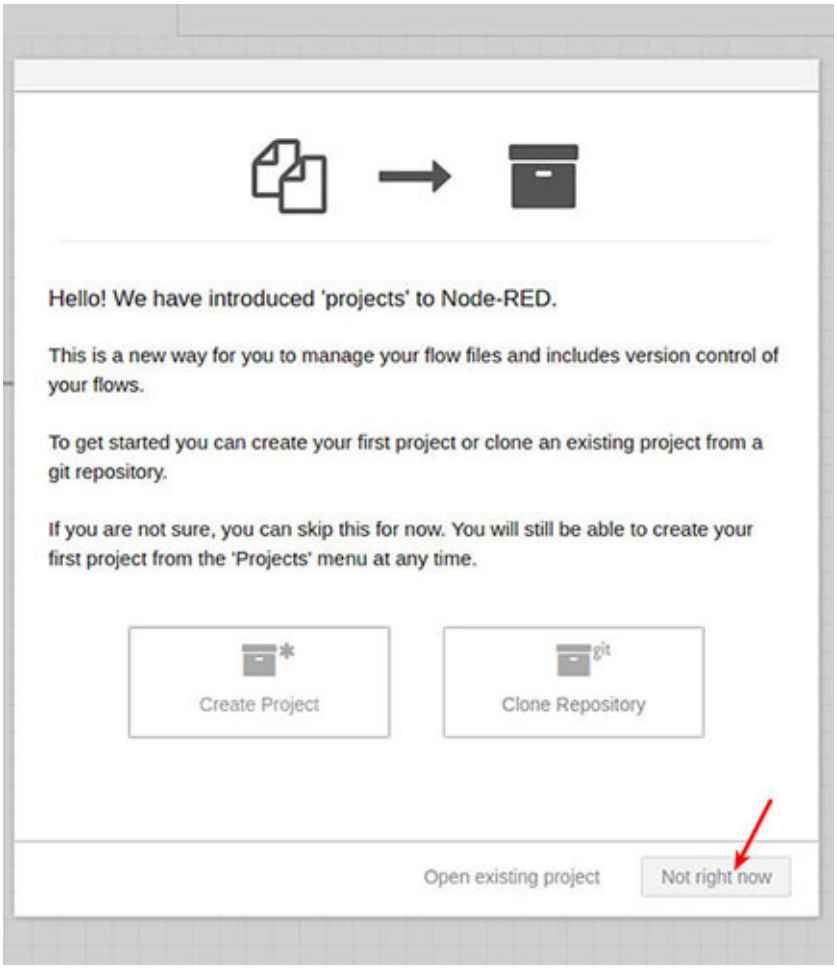


Figure 2.7: *Project set-up window*

Now, you are ready to build your first NodeRED program.

Creating your first flow

This section will guide you through the *Hello World* example, a simple digital clock updating the current time every second. The result should be as shown in the following image:

Figure 2.8: *Dashboard UI for the digital clock*

Having in mind the expected outcome, follow the steps below to do it on your own:

1. Create a new *flow* tab in your workspace by clicking the + button on the top-right part of the area. In the following screenshot, the action corresponds to the arrow labeled as step 1:

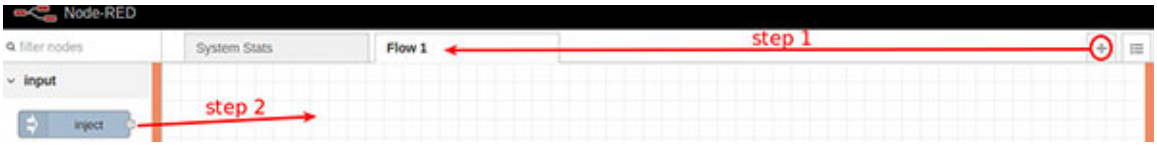


Figure 2.9: *First steps of the digital clock flow*

2. Add an `inject` type node from the Nodes palette. This type belongs to the input category listed first in the palette, as you can see in the screenshot above, looking at the origin of the arrow marked step 2. The action to do so is the classical drag and drop operation. As soon as you drop the node, its name will change to timestamp, which is the default type of object it supplies.

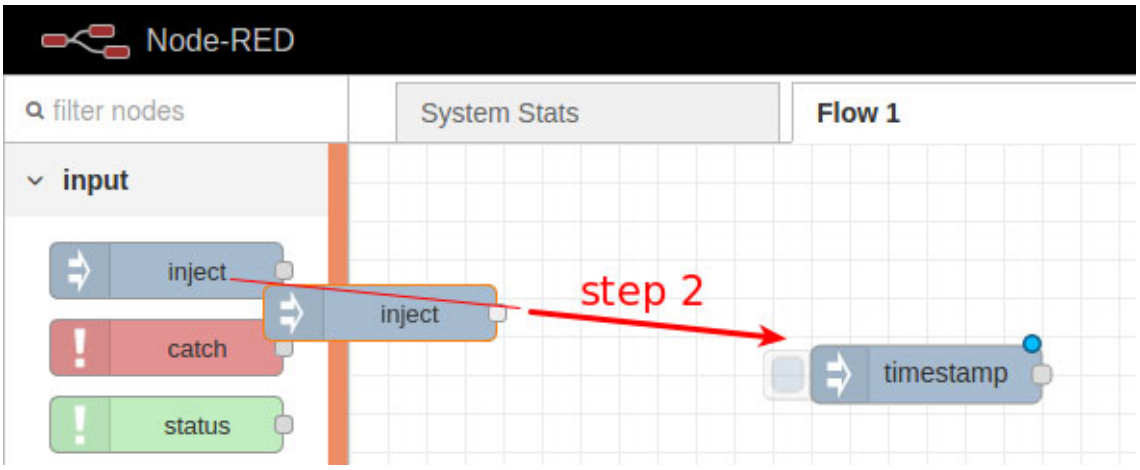


Figure 2.10: *Second step of the digital clock flow*

3. Double click on the node and its form will open letting you see a window, like the one in the next figure. You can check that there are many more types of inputs. The one we are selecting provides the UNIX timestamp of your system. Remember that the UNIX time is defined as the number of seconds since 0:00 UTC of *1 January 1970*.

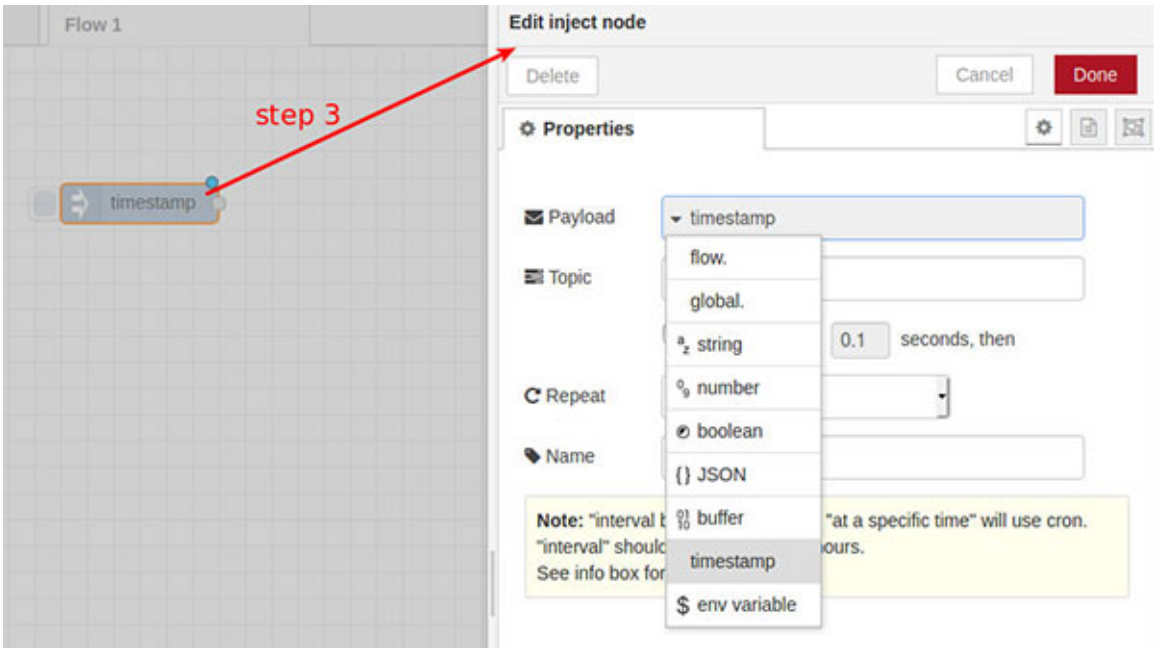


Figure 2.11: *First steps of the digital clock flow*

4. Fill in the form with the two fields indicated in the following screenshot. The first, `inject` once after, will automatically generate the first timestamp as soon as the flow starts running. The second

field, `Repeat`, has to be ticked so that a new timestamp is injected every second indefinitely, hence simulating a clock.

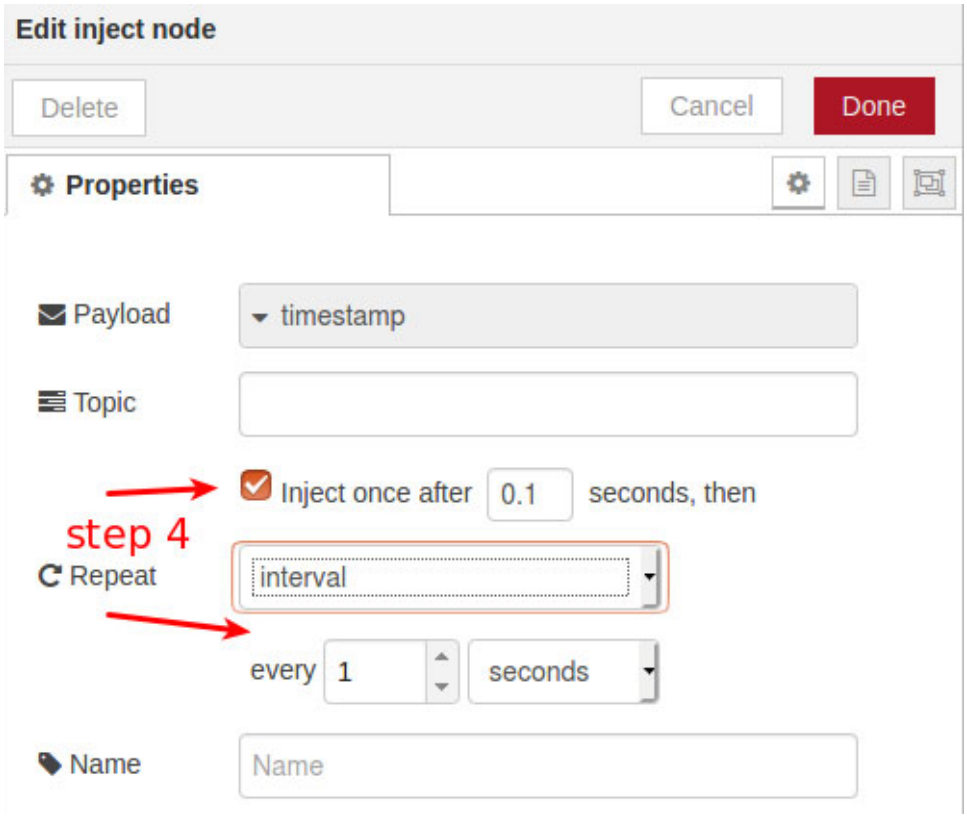


Figure 2.12: *Configuring the Inject node to send timestamps*

5. At this point, we only need an output node to check that the flow is behaving as expected. To do so, drag and drop a `Debug` type node that you can find under the output category. As soon as you drop, you will see that its name will change to `msg.payload`. As in the case of the input node, it refers to the specific data it provides. The standard name of the JSON messages that flow through the wires is 'msg', and it is composed of two fields:
    - **topic**, a string type value that tells what is the message about
    - **payload**, the content of the message, i.e., timestamp in our example

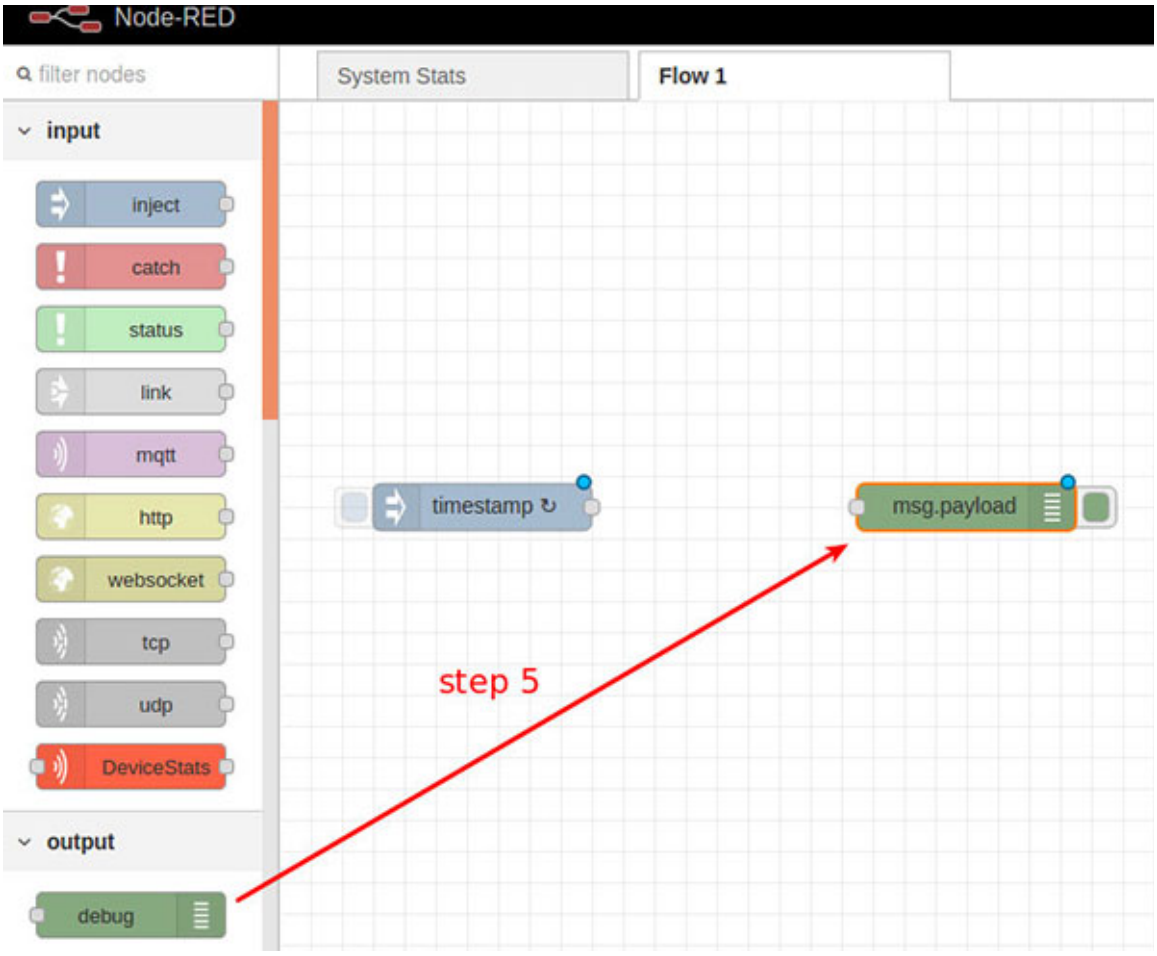


Figure 2.13: *Adding a Debug node*

6. Connect both nodes by dragging a wire between the connection points in the nodes. You can initiate the wire in either of the nodes.

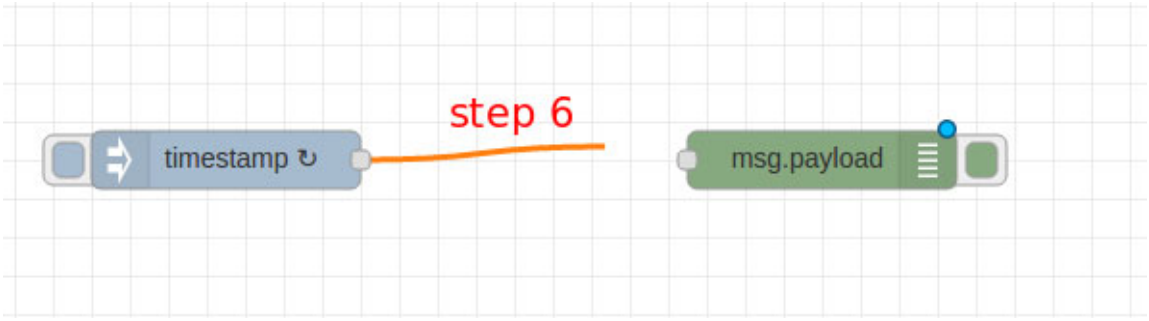


Figure 2.14: *Connecting the Debug node*

7. Once your flow is finished, you will see the `Deploy` red button in the top-right corner of the editor. By clicking it, you will make

NodeRED process your flow and start the execution. If there is no error, the button will turn grey once the flow is validated.

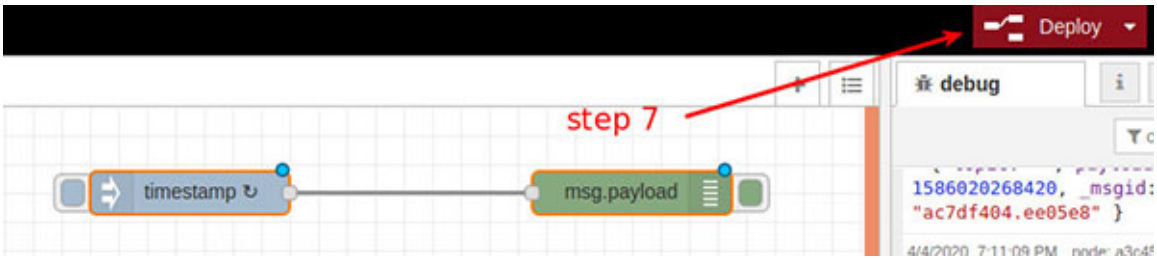


Figure 2.15: *Deploying the flow*

Note: Anytime you make any change in the flow, even if you just move the location of a node in the workspace, the Deploy button will turn red warning you that there are changes that can be transferred to the execution environment. While you do not press the button, the flow in execution will keep on being the last deployed version. Hence, if you want to restore that version, you only have to reload the browser tab of NodeRED, and changes will be discarded.

8. Once the flow is executing, visit the `debug` messages tab in the side bar by clicking on its icon. As shown in the image, make sure that only the messages of your current flow are printed. Otherwise, the output will be distorted with the output of the other flow system stats, which has been running since we launched NodeRED.

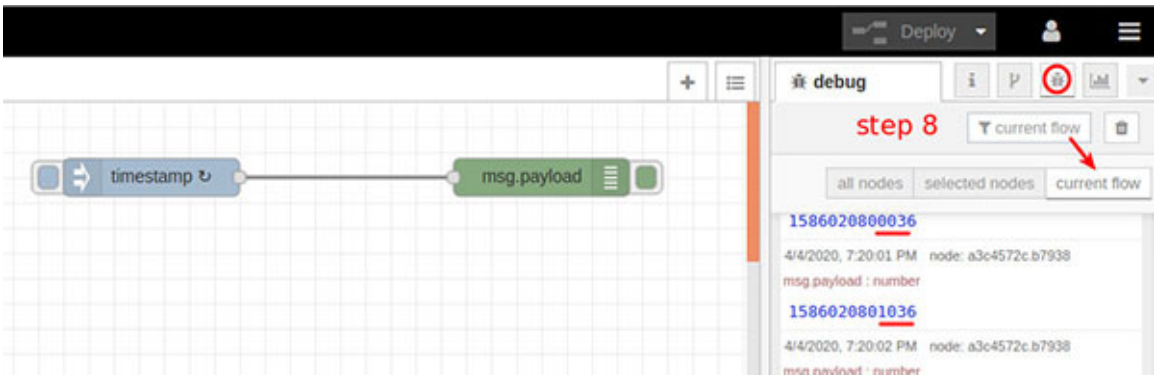


Figure 2.16: *Debug messages*

As it was pointed out in step 3 above, the inject node provides timestamps as UNIX time, i.e., the number of seconds since *0:00 UTC of 1 January 1970*. Furthermore, the time unit in JavaScript is milliseconds, which makes it even more difficult for us humans to know what time it is.

In order to check if the flow is working as expected, look at the two consecutive messages in the last picture. The timestamps finish in … **0036** and …**1036**. The difference between both is 1000 milliseconds, i.e., 1 second.

A final point to comment is the structure of the NodeRED messages that we introduced in step 5 above. In the debug window, you are only watching **`msg.payload`**, but bear in mind that the whole message is something like this:

```
{
  "topic": "",
  "payload": 1586025212841,
  "_msgid": "8f721610.1bf658"
}
```

Let's review the content of every field of the msg JSON object:

- The first field—**`topic`**—tells the type of message. It is empty at this point because we did not specify it in the input node that produces the timestamp (you can check the empty topic field of the node form in the step 4 above).
- The second field—**`payload`**—carries the UNIX timestamp in milliseconds.
- And the third field—**`msgid`**—is a unique identifier of the message to be used internally by NodeRED.

Tip: To watch complete messages, open the debug node and set the output to complete msg object, as shown in the image below:

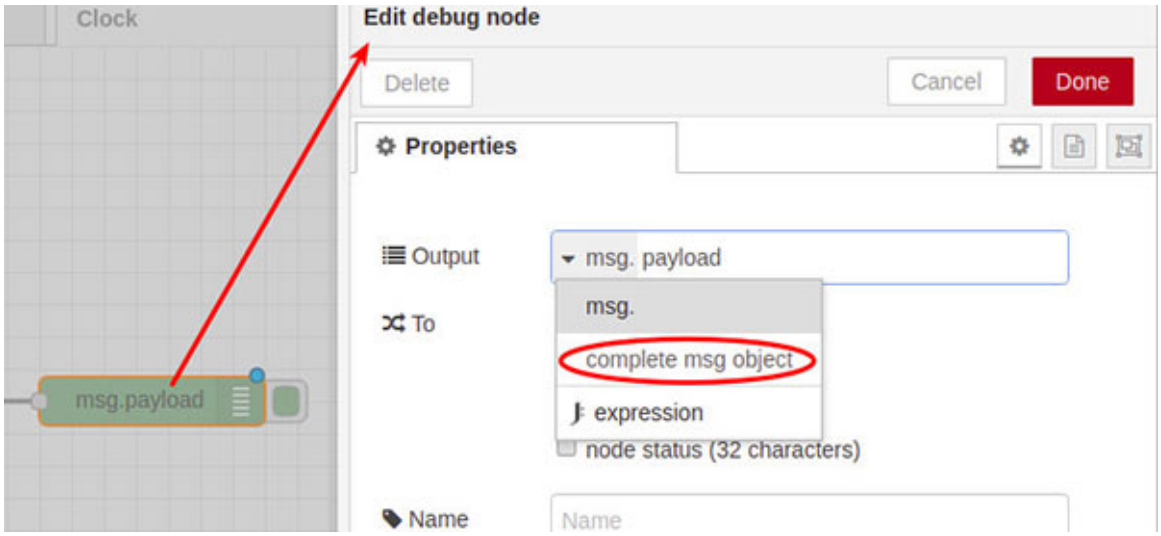


Figure 2.17: *Configure the Debug node to show the complete msg object*

In the next section, we will explain how to transform the UNIX timestamp into a human-readable time format.

Finishing the digital clock flow

We will start this section by explaining how to install contributed packages in NodeRED, something we need at this point to provide the time in a human-readable format. This is something that we can easily accomplish with node-red-contrib-moment, a NodeRED package based on the popular JavaScript library `moment.js`.

Note: Most of the contributed NodeRED packages are just wrappers of existing JavaScript modules hosted in the npm registry. Hence, many of the useful packages that the web developers need to build their web apps have also been ported to NodeRED. This means that you can include them in your project just by configuring nodes through a NodeRED form without needing to write a single line of code.Currently, there are more than 400.000 packages in npm, at present the largest software package registry in the world. Obviously, not all of them have been ported to NodeRED, but this should give you a rough idea of the rich ecosystem you have to build an IoT application quickly.

The `node-red-contrib-moment` package, whose documentation can be found at **https://flows.nodered.org/node/node-red-contrib-moment**, has many useful options. At this point, we are interested in transforming UNIX timestamps into local time, i.e. the official one in your time zone.

Installing packages

By following the installation process for `node-red-contrib-moment`, you will learn how to install any other package into NodeRED. To do so, follow the steps depicted in the following screenshot:

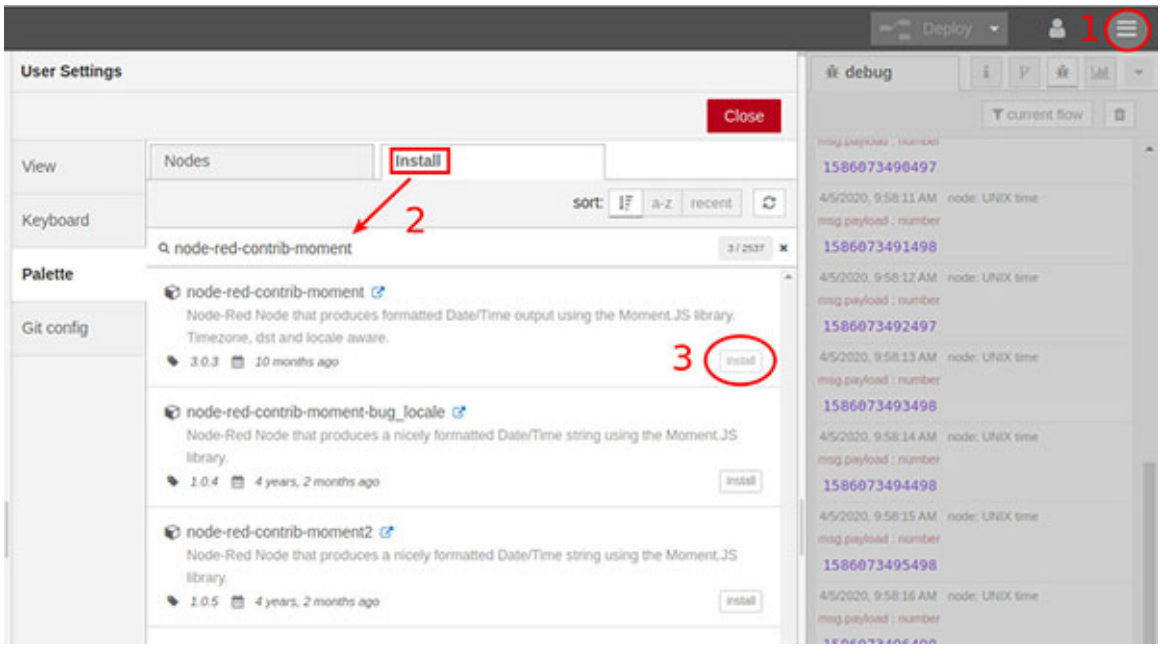


Figure 2.18: *Accessing the manage palette to install packages*

So, let's write down the steps:

1. Deploy the right menu (circle 1), click on `Manage palette`.
2. Select the `Install` tab and type in `node-red-contrib-moment` (arrow 2) to find the package.
3. Then, press the `Install` button (circle 3).
4. A pop-up window, like the one in the following image, appears with an information paragraph and three options. Confirm the installation by pressing the right button.

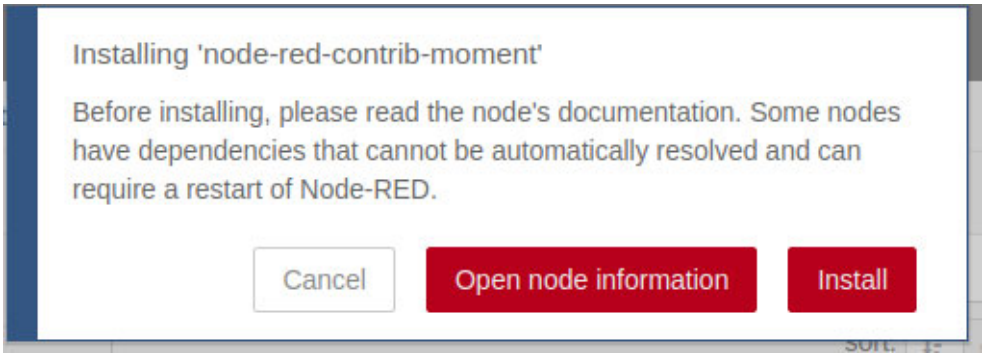


Figure 2.19: *Pop-up window for installing a package*

Once the installation is finished, look at the left pane to find the new nodes. You can locate them under the formats category: `Moment` and `Humanizer`. You can click on any of them, and see its documentation on the info tab of the side bar, as shown in the following image (the side bar has been enlarged dragging its left edge for better readability):

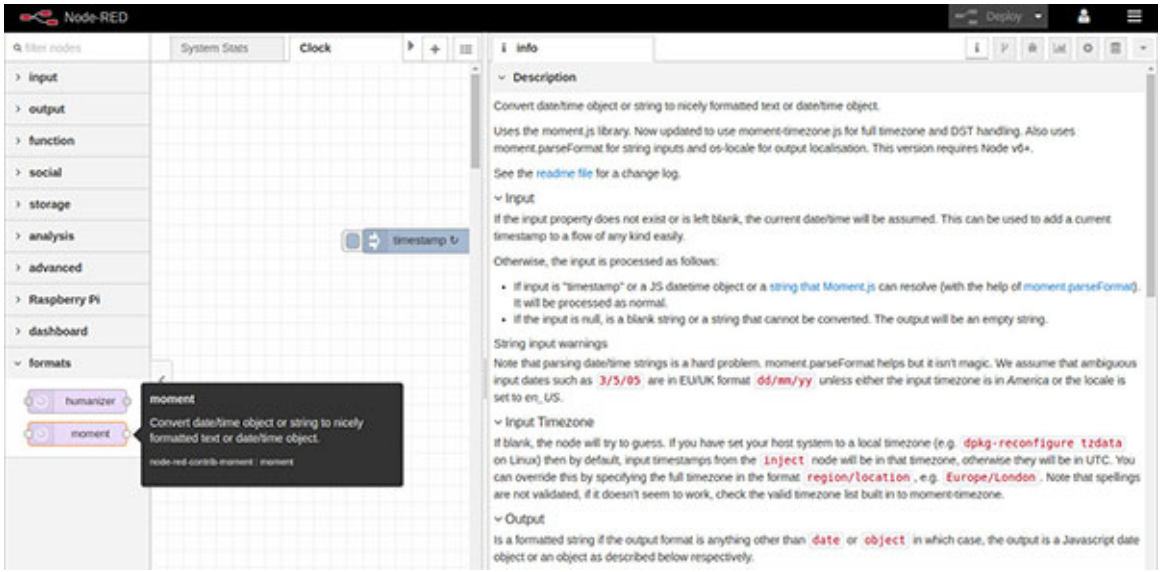


Figure 2.20: *Checking the nodes of the installed package*

At this point, we are ready to finish the first version of the flow.

Converting timestamps to local date and time

Follow these simple steps to complete your workflow design:

1. Drag and drop two `moment` nodes, since we will configure each of them with a different output format to provide the date in one and

the time in the other.

2. Add two wires, one for each of the new nodes, connecting both to the input node labeled `timestamp`.
3. Then, add two debug nodes as well as two wires to access the output of each `moment` node.
4. Give names to the four nodes, and make sure they are connected as shown in the next image:

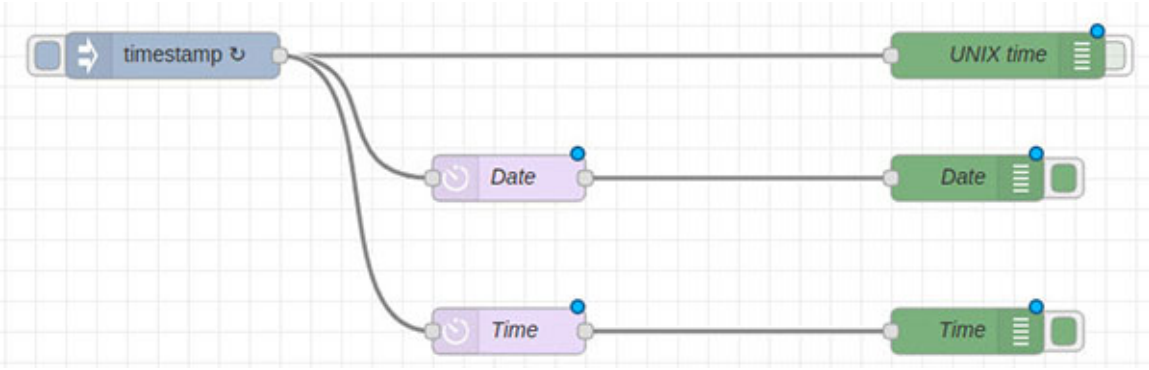


Figure 2.21: *Flow to get current date and time*

5. The node labeled `Date` provides the output calendar date, and has to be configured as shown in the following screenshot. We have marked the three fields that you have to fill in:

    - The `Input Time Zone` field [1] has to be set to **Greenwich Meridian Time** (**ETC/GMT**) because this is the time zone that has no time difference with UTC (Coordinated Universal Time), i.e., the origin for the UNIX timestamps supplied by the timestamp inject node.
    - The `Output Tz` field [2] is set to your local time zone, Europe/Madrid in the example.
    - The `Output Format` field [3] is set to day of the week (ddd), day of month (Do), month (MMM), year (YYYY).

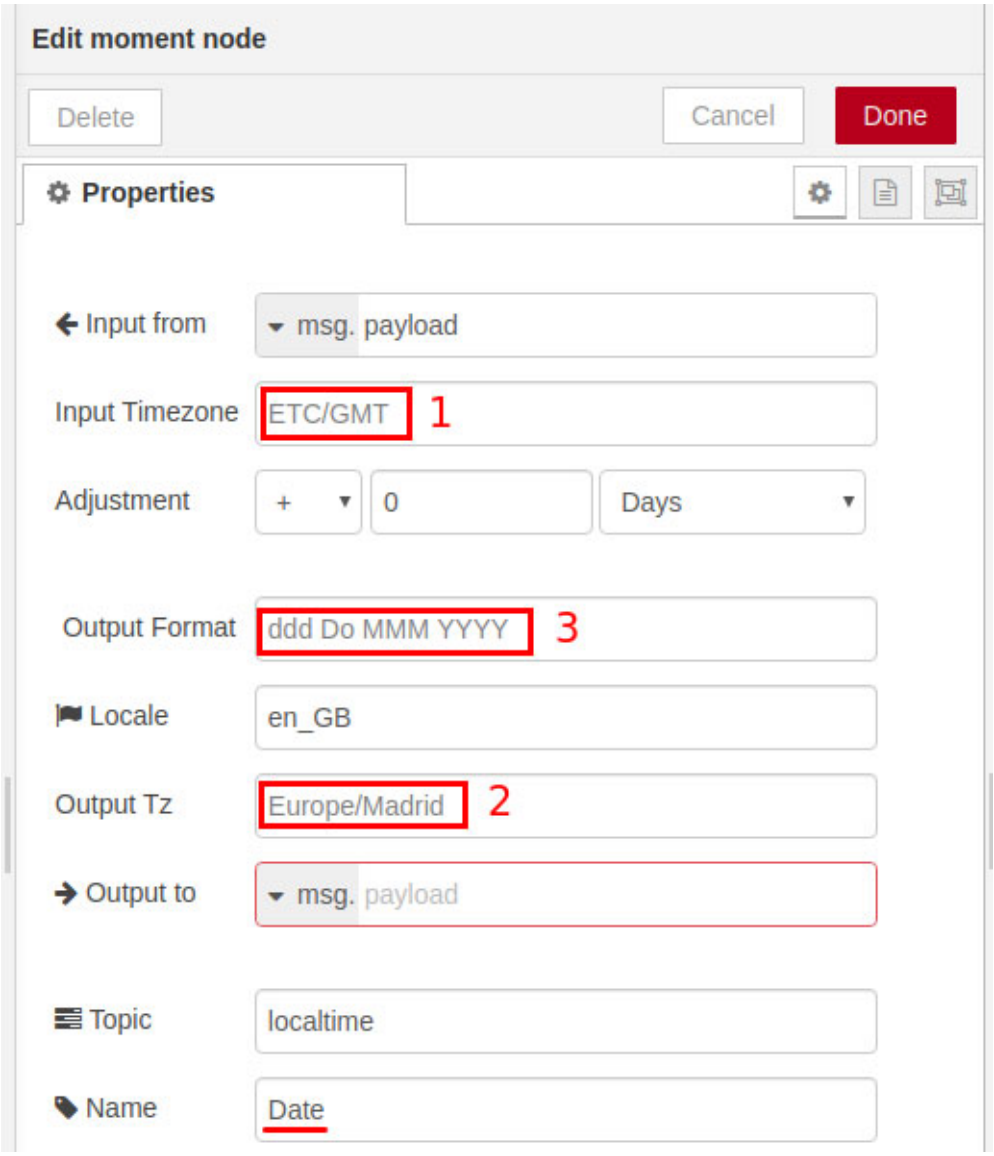


Figure 2.22: *Configuring the Moment node*

6. The node labeled `Time` is configured in the same way with the only exception that the `Output Format` field [3] has to be set to hour:minute (`HH:mm`) in order to have only the time.

Figure 2.23: *Output format of the moment node*

7. Then, press the red `Deploy` button of NodeRED and look at the `debug` tab in the side bar. By toggling on or off any of the three green Debug nodes, the corresponding messages are printed or

skipped, respectively. With this interactive feature, you can have a cleaner output and focus on the content of a specific message. In the picture below we have toggled on the `Date` and `Time` debug nodes, the ones whose output we are validating.

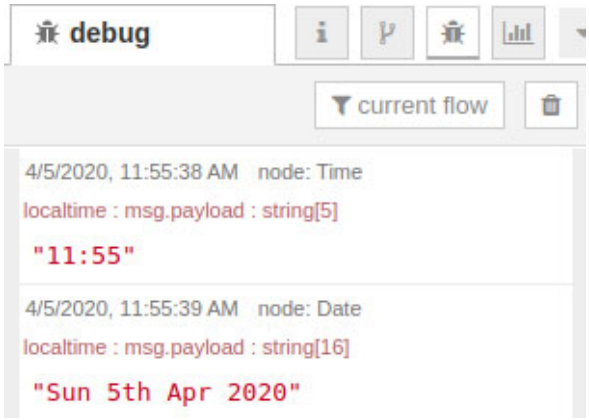


Figure 2.24: *Checking the output of the moment nodes*

At this point, we have the date and time updated every second. The last part of the example is to build a simple user interface to display this information as a digital clock.

Building the clock widget

Follow the given steps to build the UI dashboard:

1. Click on the `dashboard` icon (circle 1) of the side bar. Then, add a new tab to the `Layout` pane (arrow 2). In the form that appears when you edit the tab, type in the name `Digital Clock` (rectangle 3). Finally, press the `Update` button to save and close the form.

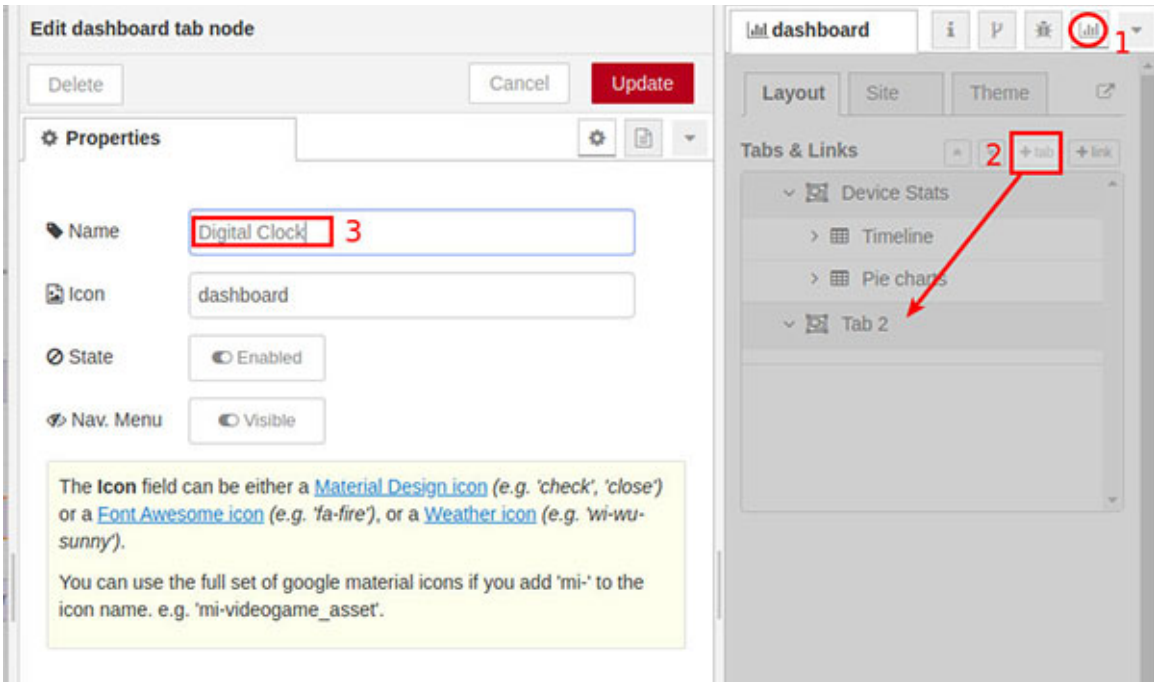


Figure 2.25: *Dashboard tab for the digital clock*

2. Add a group to the `Digital Clock` tab that you have just created. This is where you will place the widgets of your clock.

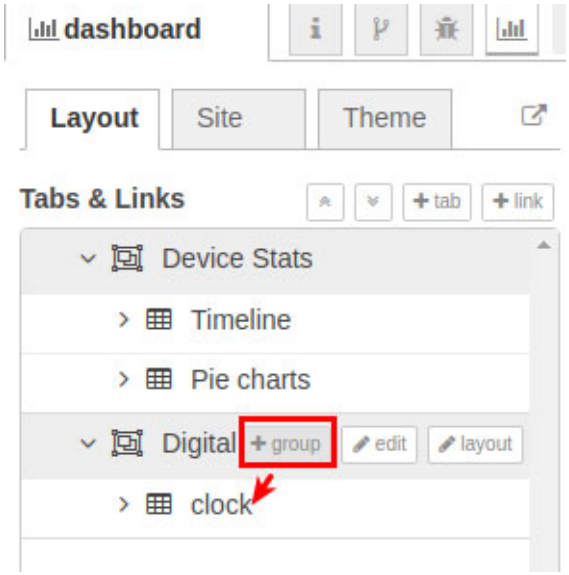


Figure 2.26: *Adding a group to the Dashboard tab*

3. Go to the `Nodes` palette, scroll down to the dashboard category, and add a `text` node to your workspace.

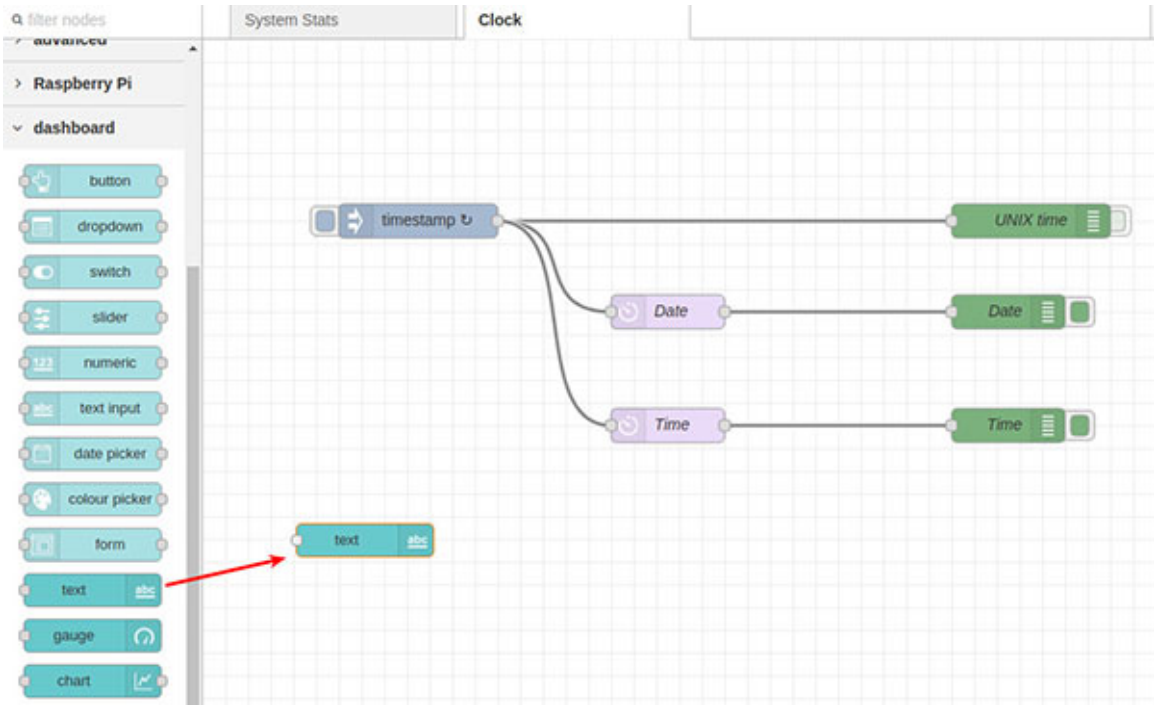


Figure 2.27: *Adding a text node for exposing the date*

4. Double click on the node and label it `Date` (rectangular red box). You can see in the `value` format field that, by default, it will take the value of the payload field of the JSON messages (underlined in red) that it receives through the incoming wire.

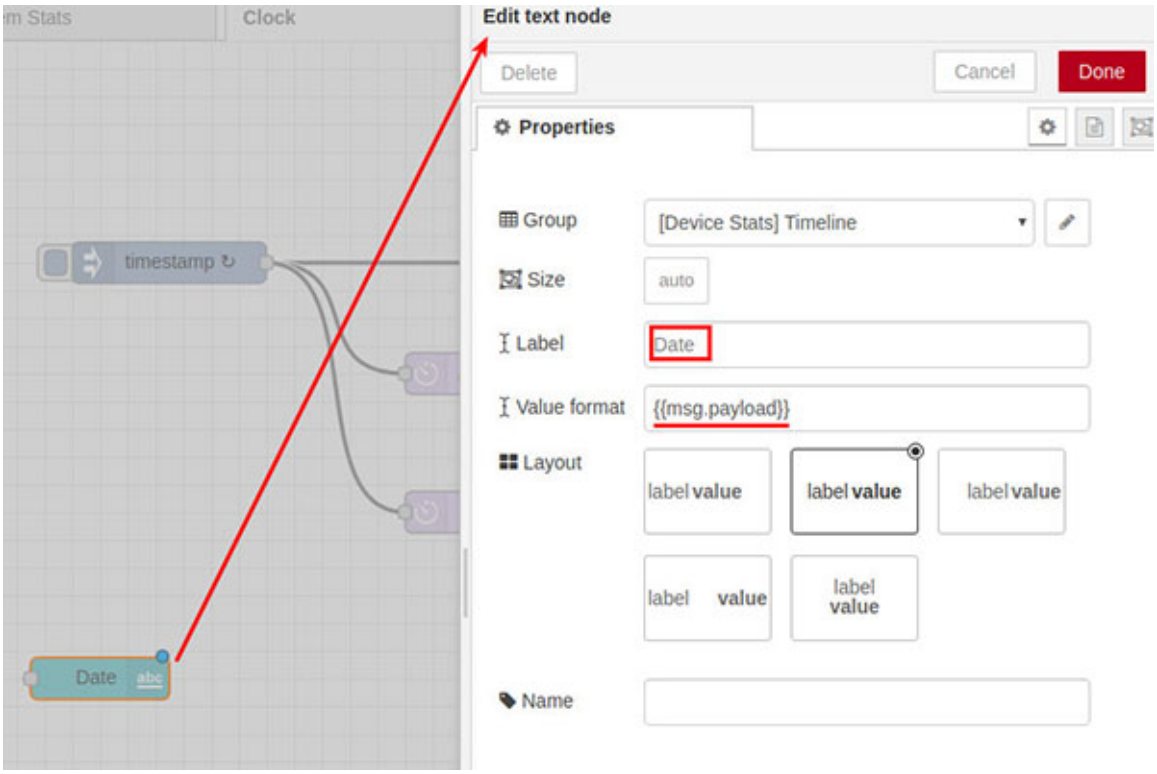


Figure 2.28: *Configuring the text node*

5. Using a wire, connect it to the `Moment` node labeled `Date`. The resulting flow should look like the one in the next image.

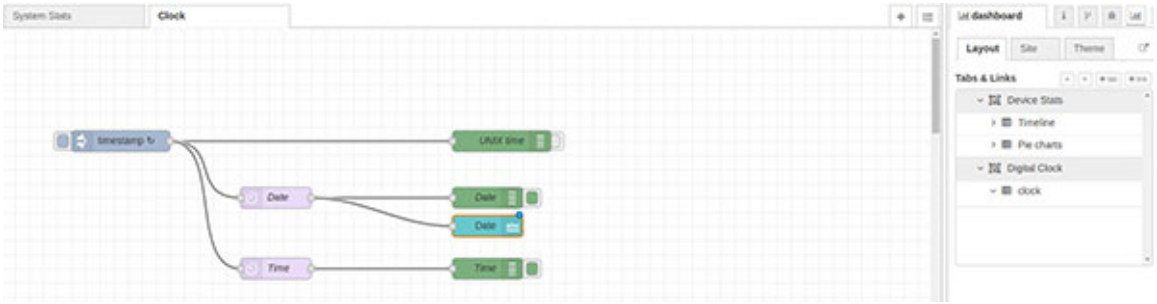


Figure 2.29: *Connecting the text node*

6. Finally, place the `dashboard` node inside the group you created in step 2 above. You can open the node, and select the group from the drop-down menu as shown in the following figure, or alternatively just drag and drop in the `dashboard` tab in the side bar.

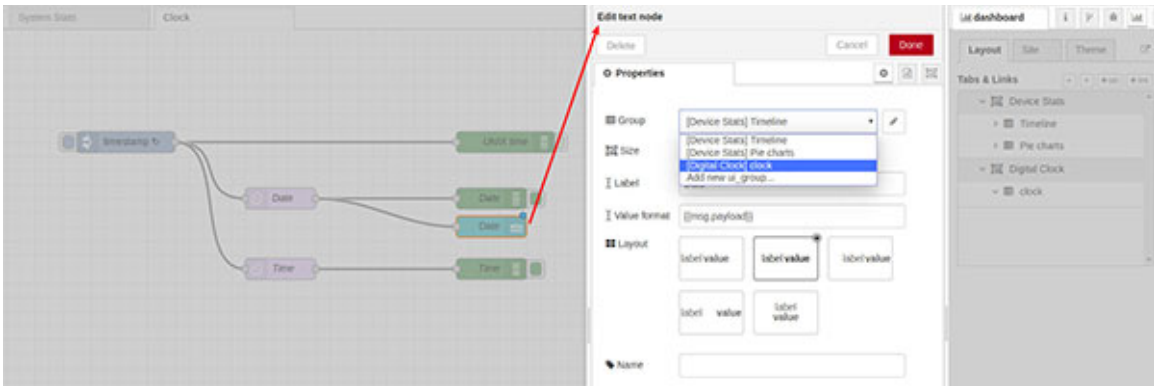


Figure 2.30: *Placing the text node inside a group of the dashboard tab*

7. Finally, press the red `Deploy` button and access the result by clicking on the link icon on the top right of the `dashboard` tab. This will open a new tab in your browser pointing to **http://localhost:1880/ui**.
8. Since there are two tabs in the Dashboard UI window, you will see in first place the `System Stats` tab we describe in the NodeRED in the brief section above. To access the clock, simply select it in the left menu and watch the result.

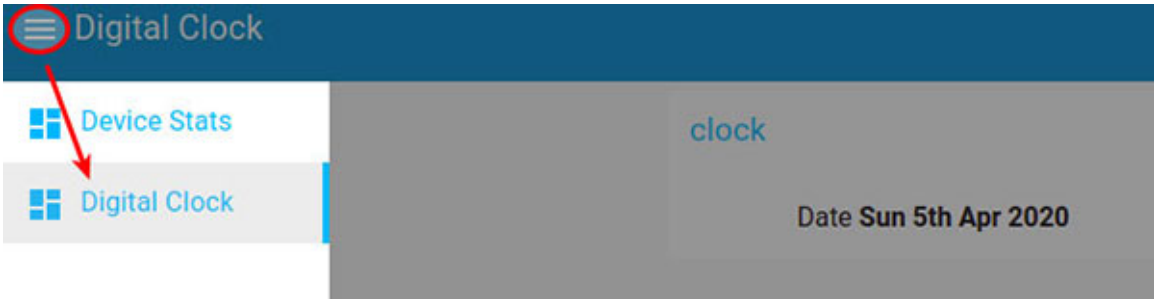


Figure 2.31: *Side menu of the Dashboard UI*

9. Finishing the dashboard to include the `Time` is as simple as copying and pasting the `Date` node and opening it and changing the label to `Time`. Finally, drag a wire connecting the timestamp input to the blue `Time` node. The resulting flow should look the same as the next image:

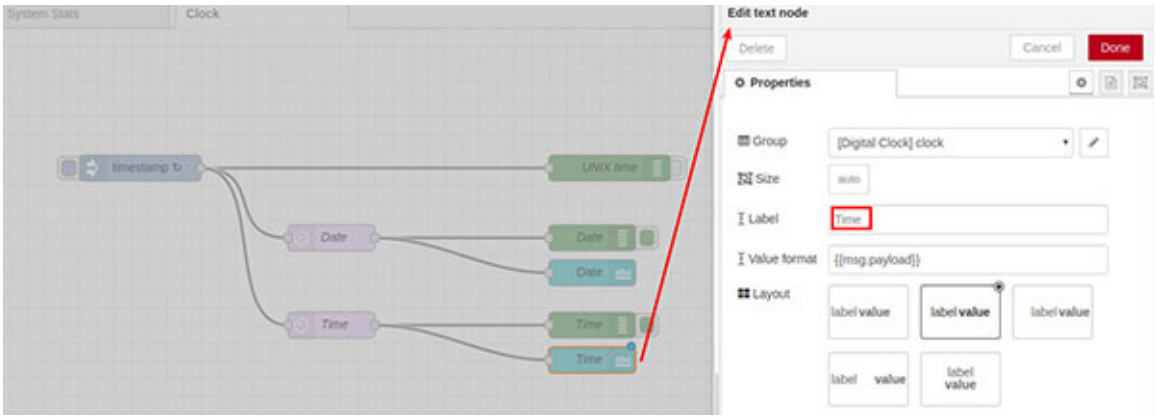


Figure 2.32: *Adding a text node for exposing time*

10. Deploy the changes and visit the **`Dashboard UI`** window in the browser to check the result.

Figure 2.33: *Visualizing time in the Dashboard UI*

11. Why does your result not look as nice as the one above? Because you still have to do some formatting in the nodes. Given below are the three modifications we need to include:

    1: Specify the font size to increase the size, i.e., <font size=6>. Do the same for the **`Time`** node.

    2: Change the layout field so that the label is on top of the value. Do the same for the **`Time`** node.

    3: Finally, in the layout of the **`Dashboard`** tab, add a spacer and place it between the two nodes.

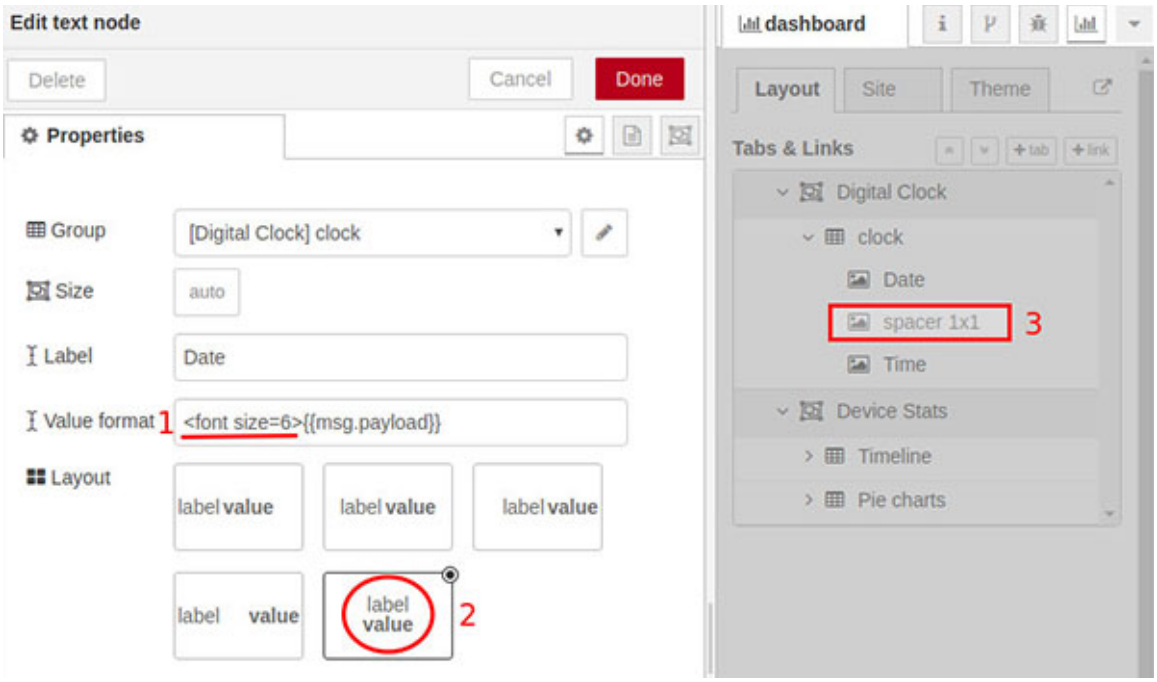


Figure 2.34: *Formatting the date in the text node*

In order to finish the example, you should compare your result with the solution provided in the repository of the chapter. That's what we will do in the last part of this section, where you will also learn how to run NodeRED with other pre-made flows.

Review the solution

Stop the application by pressing *Ctrl* + *C* from the terminal from where NodeRED was launched. Then, start again with a different command:

```
$ npm run solution
```

Automatically, NodeRED loads the solved example and you can compare it with what you did. But let's understand how we did this. Open the file `package.json`, and focus on the `script` field of the JSON object. It should look like the following snippet:

```
"scripts": {
  "start": "node node_modules/node-red/red.js --
  settings=./settings.js flows/flow.json",
  "solution": "node node_modules/node-red/red.js --
  settings=./settings.js flows/flow_solved.json"
```

```
}
```

What we run in each case with the `npm` command is NodeRED (the bold letter part in each line of the scripts) specifying a settings file, the same in both the cases, and a JSON file that contains the definition of the flow (the italics letter part in each line of the scripts). You can inspect the content of any of the files to be aware that it contains all the information that NodeRED needs to represent the flow graphically in the workspace.

Hence, to visualize the solution, the only thing we have done is to load the file `flow_solved.json` instead of the initial `flow.json`, which now also includes your work on the digital clock.

As a final thought, think how useful it is to have the flow design contained in a lightweight text file. You can copy it, move it to another flow, send via email to a collaborator, etc.

Conclusion

In this chapter, we have covered the setup of NodeRED, the core IoT component of the application that will record environmental data serving a twofold goal, i.e., provide real-time updates to the consumers (*streaming* functionality), and keep a historical record that can be retrieved any time a user has to perform specific analyses of the past (*storage* functionality).

With an easy to start example app, a digital clock, you should have acquired the essential skills to develop IoT applications with NodeRED.

In the next chapter, we will develop the first part of the project book, covering how to acquire the environmental conditions (data acquisition functionality) and stream them to the end users (streaming functionality).

Points to remember

- The Git tool is the most used in the developer community for the control version of the software.
- Node is an open source server environment-based on *Chrome v8* engine that will allow you to run the JavaScript code. It is

commonly used to run backend applications, making life easier for developers.

- NodeRED is a graphical event-driven framework to build IoT applications.
- NodeRED serves as a prototyping tool, letting you quickly sketch your ideas into a working app to show to your customers and colleagues.

Multiple choice questions

1. **What is the main purpose of the git software tool?**
    a. install software based on Node runtime environment, like NodeRED
    b. control version of the software, keeping track of changes in the code
    c. clone Github repository
    d. backup our work
2. **What is the Node environment?**
    a. a version of JavaScript to run code in a remote server
    b. the basic building block of NodeRED: inside the node we insert JavaScript code
    c. an open source server environment-based on Chrome v8 engine that will allow you to run JavaScript code
    d. a synonym of the JavaScript programming language
3. **What is the concept of flow in NodeRED?**
    a. the set of wires that we use to connect several nodes in the workspace
    b. the set of nodes and wires connecting them in a single tab of the workspace
    c. the feed of messages that travel through the wires and nodes
    d. the analogous of a fluid in a hydraulic system of pipes
4. **How is a NodeRED flow physically stored?**

a. in a text file saved to the hard disk of the PC
b. in a binary file saved to the hard disk of the PC
c. in the database of NodeRED
d. only in the RAM while NodeRED is running

5. **What is the purpose of the package node-red-dashboard?**
   a. to provide the means to permanently store the flows in the hard disk
   b. to provide an administration user interface for the developer
   c. to avoid the need of using the debug tab of the Side bar for testing a flow
   d. to provide a friendly user interface to graphically show the results

Answers

1. **a**
2. **c**
3. **b**
4. **b**
5. **d**

Questions

1. What is JavaScript?
2. What is Node?
3. What is NodeRED?
4. What is the purpose of the data acquisition functionality built with NodeRED?

Key terms

- **JavaScript**: the programming language of the web, used to run scripts in pages loaded in the browser.

- **Node**: an open-source server environment to write full applications using JavaScript, without necessarily needing to make use of a web browser.
- **NodeRED**: a graphical event-driven framework to build IoT applications.

Part - II

Getting Familiar with Software Toolkit

CHAPTER 3

Data Acquisition and Real-time Streaming

Introduction

This chapter covers the IoT edge layer and the entry point to the streaming API (Transportation layer). So, it covers the whole IoT foundation layer. It explains how to perform data acquisition from the sensors and transmit them via the streaming API. The code that we will build in this chapter will run entirely on the IoT device.

Starting from the simple *digital clock* app of *Chapter 2, Getting Started with NodeRED*, we will develop another app that will acquire the environmental conditions from a sensing device called *Sense Hat,* and stream them to make available for real-time processing.

To do so, we will cover several steps: get familiar with the virtual Sense Hat board, explain how to create a NodeRED project, integrate the external streaming tool called *Pusher*, and finally learn to store in NodeRED the current readings of the Sense Hat.

This chapter will provide you with the foundation to create an application by integrating virtual sensors and a streaming app within the environment of NodeRED.

Structure

In this chapter, we will discuss the following topics:

- Getting to know the Sense Hat
- Real-time apps with Pusher
- Publishing environmental data to a Pusher channel
- Saving system state in NodeRED variables

Objectives

After studying this unit, you will learn how to acquire data from commonly used sensors and produce real-time updates by publishing data using streaming apps. Furthermore, you will learn more about two NodeRED advanced features: run different applications in the same instance of NodeRED using the Projects feature, and persist the application state in the physical storage, to make it resilient to restarts, crashes, or any other event that might clear the dynamic memory (RAM).

Technical requirements

The code for this chapter is in the GitHub repository that you can find at **https://github.com/Hands-on-IoT-Programming/chapter3**

In order to get your local copy, go to your home path and clone the repository as follows:

```
$ cd ~/book_hands-on-iot
$ git clone https://github.com/Hands-on-IoT-
Programming/chapter3.git
```

Hence, change to the path of the cloned folder in order to watch the contents of the file:

```
$ cd ~/book_hands-on-iot/chapter3
$ ls -la
```

There is a complimentary repository for this chapter regarding the topic of NodeRED projects that will be explained in the next section. Its URL is:

https://github.com/Hands-on-IoT-Programming/chapter3-sense-hat-simulator

You don't need to clone it now, since this step will be part of the project feature explanation, and will be retrieved straight from NodeRED. Its reference is included here in order to provide a complete overview of the code of the chapter.

Since for all the chapters in this *Part II Getting familiar with the software toolkit* we will put the focus in the software, both the *Raspberry Pi* and the *Sense Hat* will be emulated. For the Raspberry Pi, we already did it in the previous chapter using the laptop, and for the Sense Hat, we will perform the simulation using a NodeRED contributed package.

In any case, we need to start the chapter by acquiring the background of what the Sense Hat board is, and what kind of data it will provide for your IoT application.

Getting to know the Sense Hat

The Sense Hat is a general-purpose add-on board for the Raspberry Pi intended to experiment with several common sensors without dealing with the complexity of wiring each of them. You just have to plug the board on top of the Raspberry Pi and connect through its **General Purpose Input Output** (**GPIO**) pins, as shown in the top part of the following figure, where you can see both boards assembled and working, i.e., LED matrix shining:

Figure 3.1: *The Sense Hat board plugged on top of Raspberry Pi (image courtesy:* ***https://commons.wikimedia.org/wiki/File:Raspberry_Pi_with_Sense_HAT.jpg*** *License CC-BY-SA 4.0)*

Tip: The Raspberry Pi website provides a very nice tutorial to get familiar with the Sense Hat. You can find it at **https://projects.raspberrypi.org/en/projects/getting-started-with-the-sense-hat**

The Sense Hat has the following elements:

- temperature sensor
- humidity sensor
- pressure sensor
- LED matrix 8x8 (= 64 color LEDs)
- joystick to produce four kinds of events: up, down, left, and right
- accelerometer

- gyroscope
- magnetometer

The set of accelerometer, gyroscope, and magnetometer is commonly known as **Inertial Measurement Unit** (**IMU**), and it is typically used in aircrafts, satellites, and drones to drive its motion according to a planned path. The IMU provides position and orientation data in real time, so that motors can continuously adjust their thrust to keep following the desired path.

For the IMU, it is crucial to take into account what the direction of the axes are. For the case of the Sense Hat, the next figure shows their orientation:

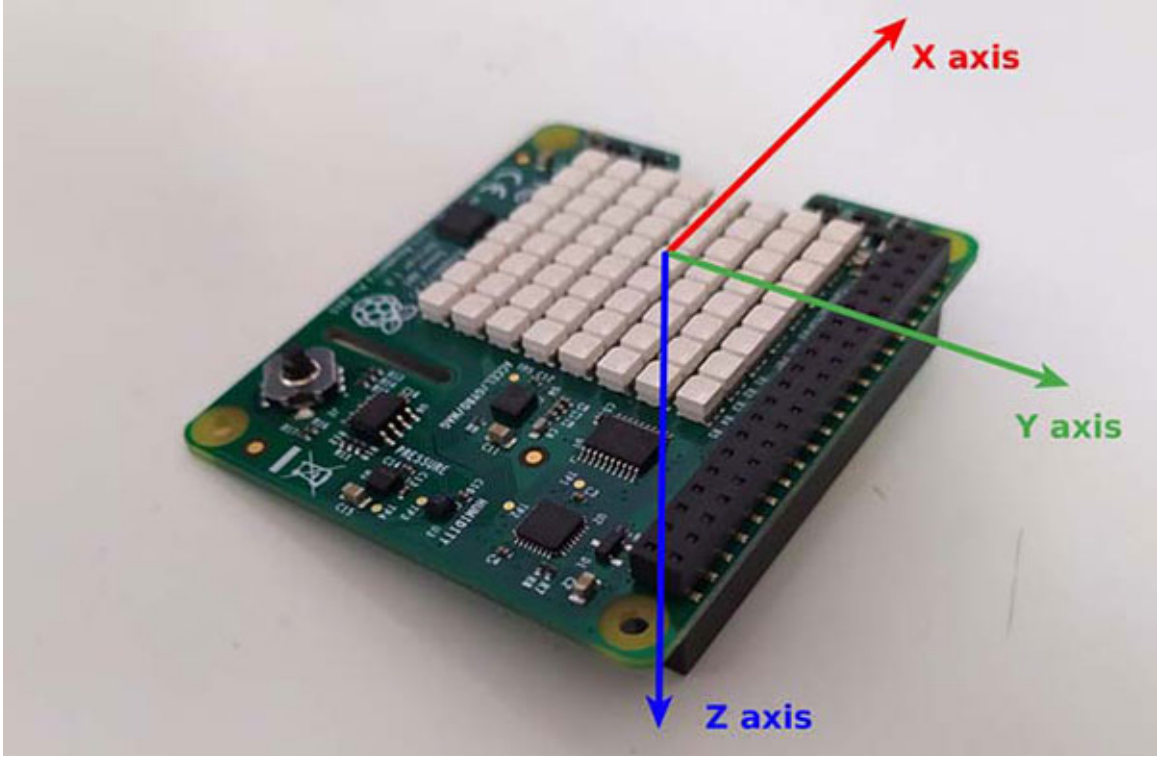


Figure 3.2: *Frame of reference for Sense Hat*

We will describe the sensors from the point of view of the events that NodeRED will be receiving:

1. **Motion events**: They correspond to the IMU since they are related to the way in which the device is moving and changing its orientation.

a. **Acceleration.x/y/z**: The accelerometer measures the acceleration in every axis of reference. The values are given in g's, i.e., the standard gravity acceleration at sea level, whose value is 9.81 m/s². Hence, if you place the Sense Hat on top of a table, you will receive 1 g along the Z axis, and 0 for both X and Y axes.
b. **Gyroscope.x/y/z**: The gyroscope measures the rotation speed about every axis X, Y, Z.
c. **Orientation.roll/pitch/yaw:** It is provided as a set of 3 values: roll corresponds to the rotation about the X axis, pitch about Y and yaw about Z. They stand for the orientation in space and are obtained as a mathematical computation from the data provided by the accelerometer and the gyroscope.
d. **Compass**: The magnetometer measures the direction of the earth's magnetic field in the same set of axes.

2. **Environment events** groups the measurement of the temperature, humidity and pressure sensor.
    a. **Temperature** measurement is in Celsius degrees (ºC).
    b. **Humidity** is expressed in %, measuring its relative value.
    c. **Pressure** is measured in millibars, being 1013 mb the standard pressure at sea level.

For the scope of this chapter, and all in part II, we will deal only with *environmental events*, which are simpler to understand and treat. We left the motion events for *Part III, Hands on IoT programming*, i.e., the last two chapters of the book.

Quick start for simulating the Sense Hat

Before getting again into NodeRED, we introduce in this subsection a standalone board simulator called *Sense Emu*, whose documentation is at the URL **https://sense-emu.readthedocs.io/en/v1.0/sense_emu_gui.html**. Sense Emu provides a nice GUI to explore the Sense Hat and to perform simple simulations using Python as the programming language.

Tip: Although in this book we are intentionally avoiding written code in favor of visual programming, you should be aware that Python is the typical beginners' language, and it is the common choice to make applications for Raspberry Pi. So, it is recommended that you get used to reading Python snippets. This language is easily readable for humans and provides a simple syntax that helps to follow a fast learning curve.

Install Sense Emu following the next steps in a bash terminal:

1. Add the repository source:

```
$ sudo add-apt-repository ppa://waveform/ppa
```

2. Update the system so that the new source is available to install software:

```
$ sudo apt-get update
```

3. Finally, install Sense Emu packages:

```
$ sudo apt-get install python-sense-emu python3-sense-emu
sense-emu-tools
```

This operation also installs a minimal Python IDE called *idle*.

Once finished, we are going to run several Python scripts to see the Sense Hat in action. All these scripts are inside the folder `sense-emu` of the repository of this chapter:

```
$ cd ~/book_hands-on-iot/chapter3/sense-emu
```

List the files in the folder:

```
$ ls -la
-rwxrwxr-x 1 ubuntu ubuntu 2275 abr 25 12:56 bar_graph.py
-rwxrwxr-x 1 ubuntu ubuntu 300 abr 25 12:56 humidity.py
-rwxrwxr-x 1 ubuntu ubuntu 783 abr 25 12:56 joystick_events.py
-rwxrwxr-x 1 ubuntu ubuntu 1248 abr 25 12:56 rainbow.py
-rwxrwxr-x 1 ubuntu ubuntu 2454 abr 25 12:56 sensor_menu.py
-rwxrwxr-x 1 ubuntu ubuntu 231 abr 25 12:56 temperature.py
```

The `x` stands for the execution permissions of the files. The first `x` is for the user (`ubuntu` in the example, in bold letters), the second `x` for the group that the user belongs to (also `ubuntu`, in italics letters), and the third for every other user in the system. Strictly, we only need the first x, since we will be executing the scripts logged in the system with the user `ubuntu`.

Note: Those permission comes with the files from the repository you cloned. If, for some reason, they are not as expected, run the following command to get the files to be executable: `$ chmod +x *.py`

Every script simulates a single feature of the Sense Hat. Let's start with the temperature by running the following Python code:

```
$ ./temperature.py
```

A desktop window like the one shown in the next image will be launched. We formally call this window a GUI, the acronym of Graphical User Interface. The slider in the right side allows the user to manually set a temperature. The response of the Sense Hat will consist of illuminating in red color more LEDs of the matrix (progressing to its bottom edge) as the temperature is increased. The left part of the image below shows the state for 27.2 ºC, while in the right part we increased the temperature up to 49.8 ºC.

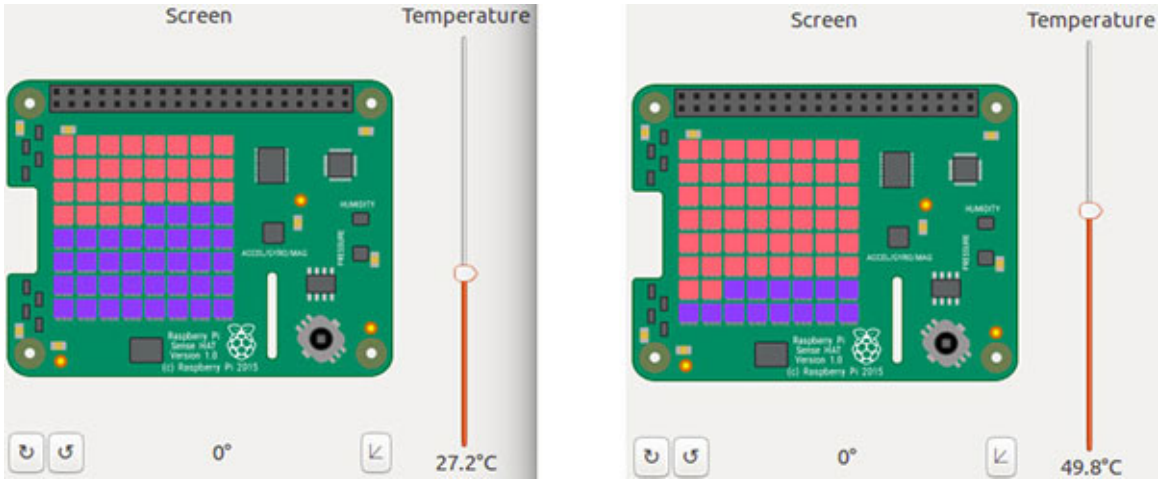


Figure 3.3: *GUI of the Sense Emu app at two different temperatures*

Note: This simulation is very simple, since it is the user who manually sets the temperature that the board will report. In the physical Sense Hat, the temperature sensor integrated in the board measures the ambient temperature and hence reports an actual value. The same happens with the rest of variables we are going to explore below, i.e. humidity and pressure.

For the **humidity** emulation, run the corresponding script (press *Ctrl* + *C* to stop the temperature emulation and gain access to the terminal):

```
$ ./humidity.py
```

You will have access to the humidity slider in the GUI, and will be able to modify the value as you wish.

There is a third script to control at the same time the temperature, humidity, and pressure with their respective sliders:

```
$ ./bar_graph.py
```

In this case, the program for the LED matrix is modified to visually report the variable levels with three bars, one for each as shown in the following image:

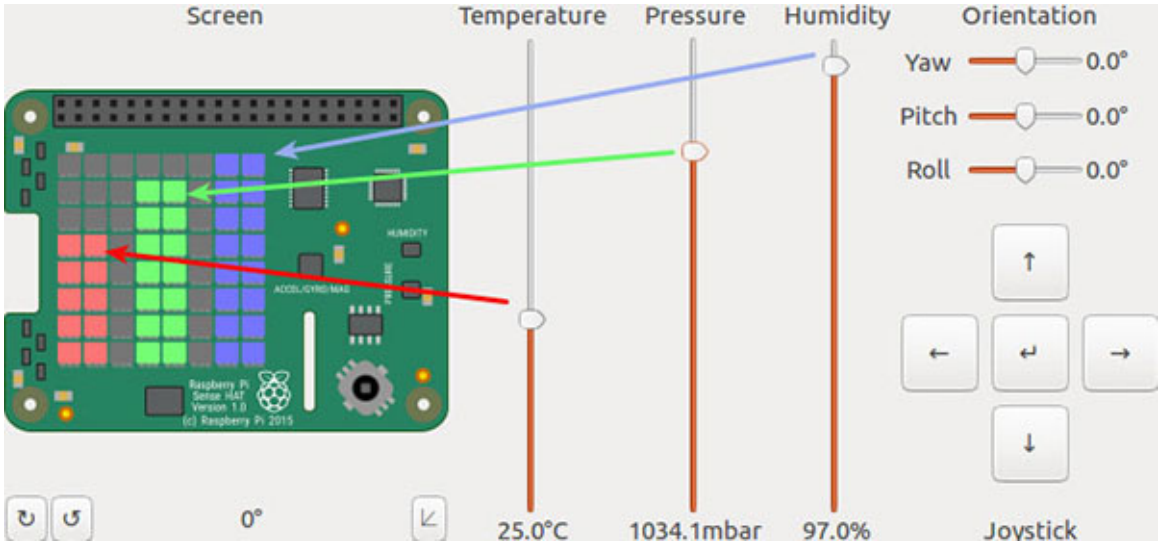


Figure 3.4: *GUI for the bar_graph.py script*

A more complete script introduces the use of the joystick to select the variable you want to show the value in the LED matrix. Execute it as

follows:

```
$ ./sensor_menu.py
```

Select the red T for the temperature, the green P for the pressure, or the H for the humidity. The fourth item, Q, allows you to quit the program. The following picture shows the corresponding GUI:

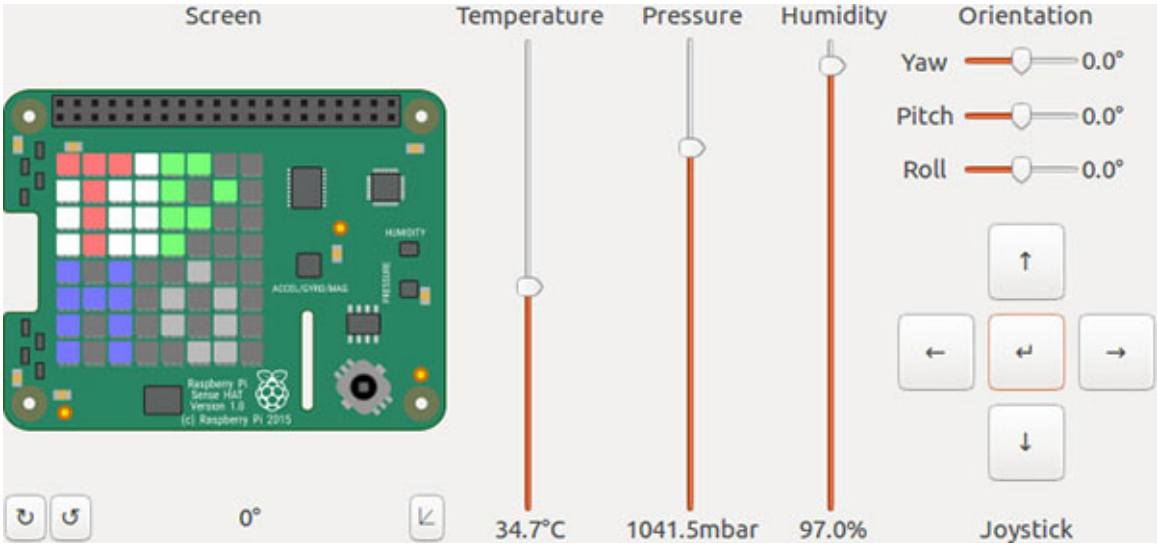


Figure 3.5: *GUI for the sensor_menu.py script*

Move the selector in the LED matrix (the white square in the background whose size is ¼ of the matrix, i.e., 4x4 pixels) using the arrows of the joystick, and press the central button to select the highlighted variable. Once you do so, the current value of the selected variable will be shown in the LED matrix as a text string that scrolls from right to left. The reported value will be shown in the slider.

To understand the events of the joystick, a script is provided to see how it works in practice:

```
$ ./joystick_events.py
```

A white pixel will move one position to up, down, left or right, according to the arrow key you press, as it is shown in the next picture:

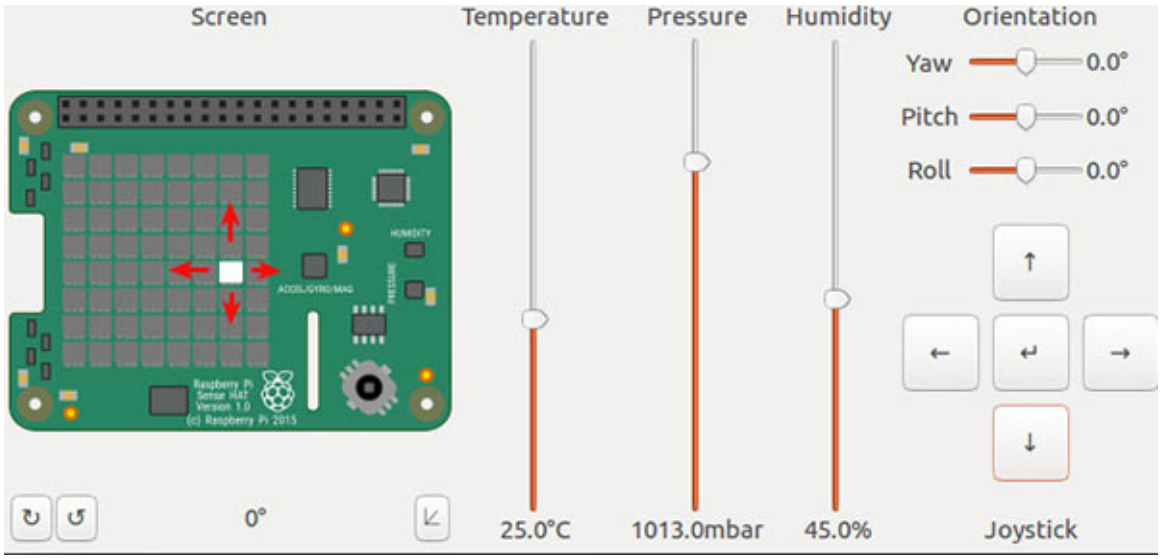


Figure 3.6: *GUI for the joystick_events.py script*

To finish this introductory subsection, you can see a nice effect by running the following script:

```
$ ./rainbow.py
```

As shown in the next image, you will see a rainbow whose line of colors translate diagonally from the right-bottom part of the matrix to the left-upper corner:

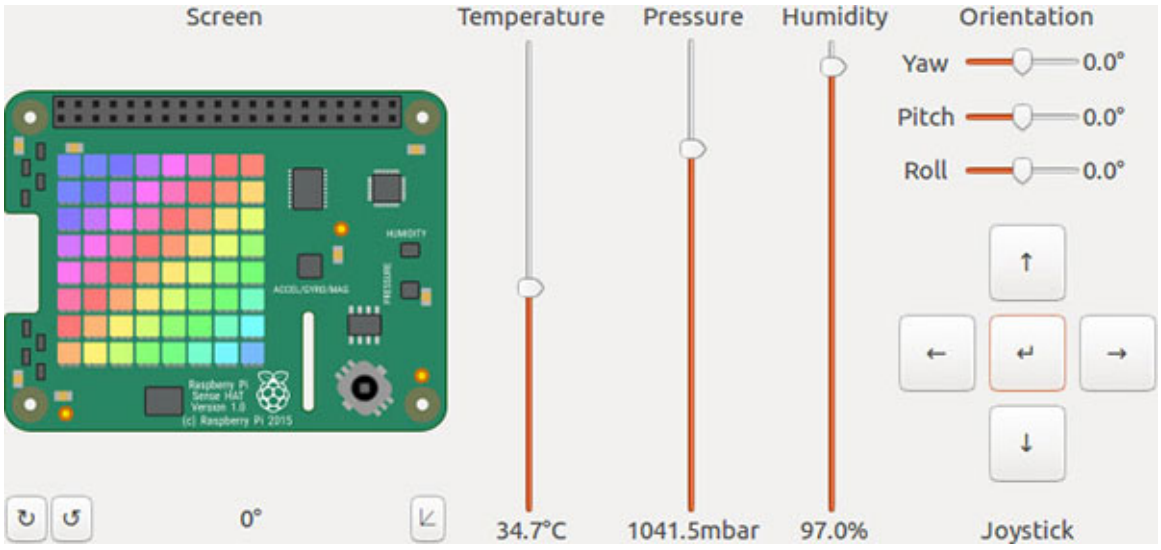


Figure 3.7: *GUI for the rainbow.py script*

Tip: You can deep dive into more commands of the simulator by reviewing the documentation at **https://sense-emu.readthedocs.io/en/v1.1/sense_emu_gui.html**

After this quick introduction to get familiar with the sensors board, let us explore the option to integrate a simulated Sense Hat right into NodeRED.

Setting up a virtual Sense Hat in NodeRED

To integrate a simulated Sense Hat into NodeRED, there is a specific package at the npm registry that provides such a simulator for this Raspberry Pi add-on board. Let's see how to install and make it work:

1. Following the same steps described in the **NodeRED in brief** section, install the dependencies and then launch the app:

```
$ cd ~/book_hands-on-iot/chapter3/app
$ npm install
$ npm start
```

2. Once the NodeRED flow is deployed, you can access the editor **User Interface** (**UI**) at the URL `http://localhost:1880`
3. You will be welcome again with the already familiar System stats flow. Log into the app with the user ID and password we configured in the previous chapter (user 'admin' with password 'raspberry').
4. Add a new tab to the workspace and you will be presented with a blank canvas. You can perform this action by pressing the + sign in the top-right part of the workspace.
5. Before placing any node, you need to install the required package following the sequence `Manage palette` | `Install` | `node-red-node-pi-sense-hat-simulator`, which is visually depicted in the following screenshot. The step-by-step process is as follows:
    - Deploy the right menu (circle).
    - Click on `Manage palette`.
    - Select the `Install` tab (arrow).

- Type in `sense-hat-simulator` to find the package.
- Then, press the `Install` button of the first of the listed projects (underlined).

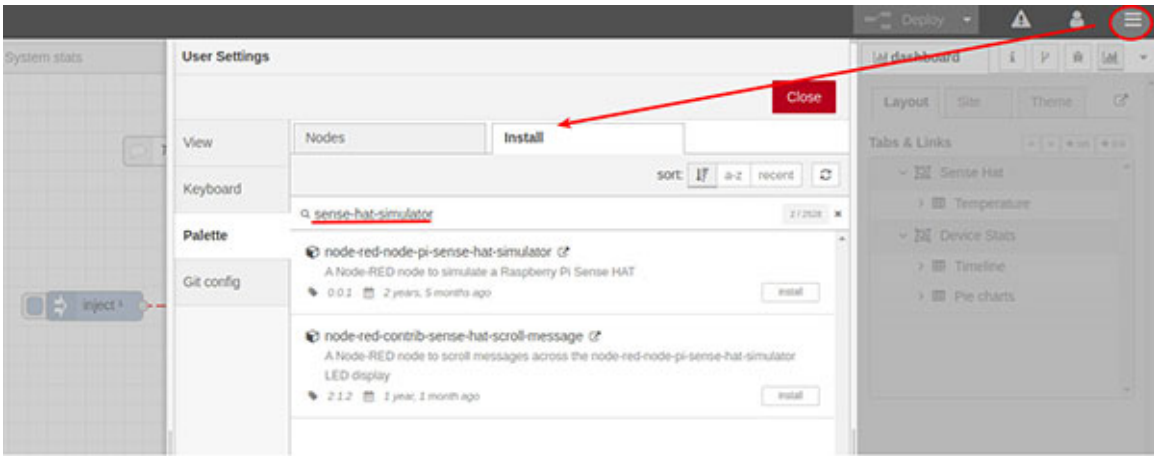


Figure 3.8: *Installing the sense-hat-simulator package*

You can find the documentation for the package in its NodeRED page at **https://flows.nodered.org/node/node-red-node-pi-sense-hat-simulator**.

6. Go to the Nodes palette (in the left pane), and type in `sensehat` to find the Sense Hat simulator nodes. You will find an input node (point of connection on the right edge of the box), and an output node (connector on the left edge). The input node produces sensor data, while the output node provides the readings of the sensor. Wire the nodes as shown in the following screenshot:

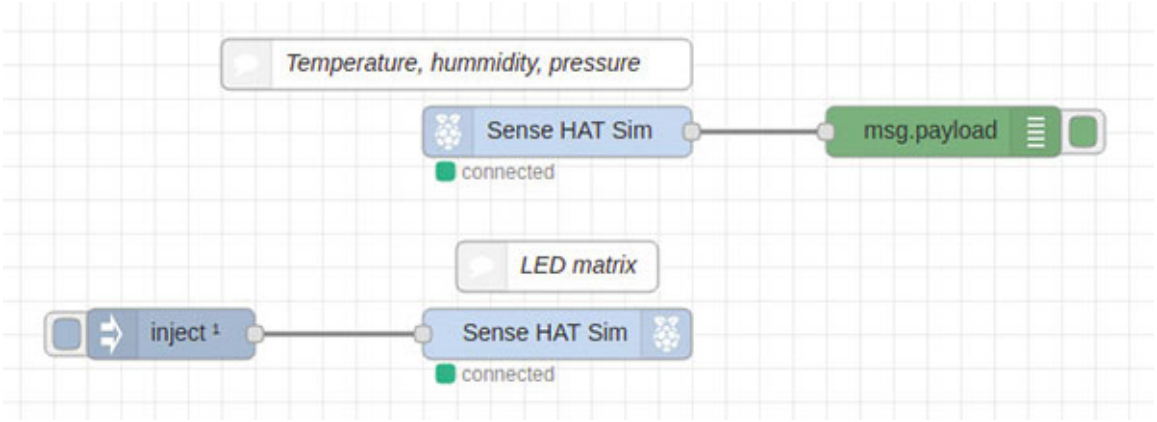


Figure 3.9: *Basic flow for the Sense Hat*

7. Find the virtual sensor at the URL **http://localhost:1880/sensehat-simulator**, of which you can see a screenshot in the image below. You can manually modify the values of temperature, humidity, and pressure, as well as send joystick strokes. Hence, this is a manual simulator where the user can interactively modify the conditions as he wishes by pushing the +/- buttons of each variable.

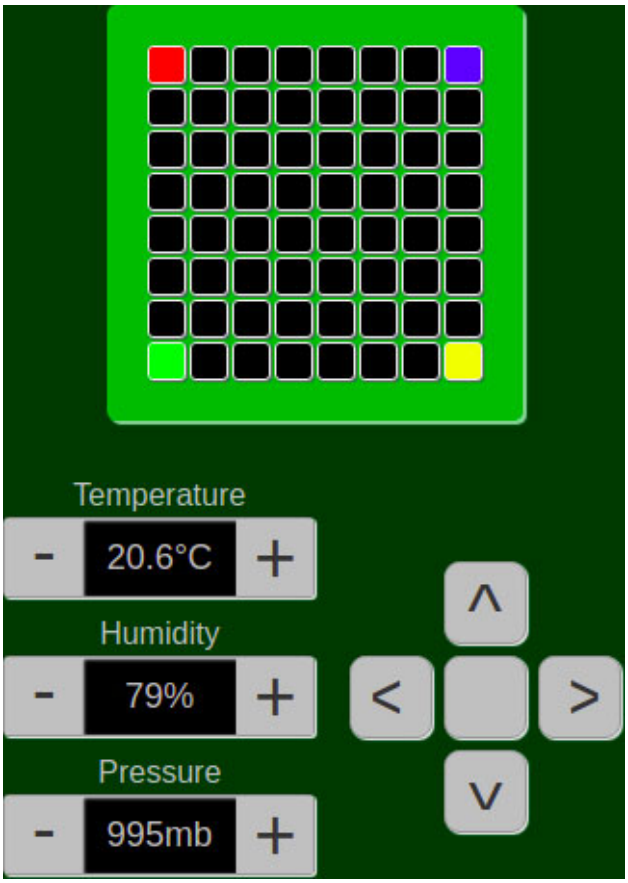


Figure 3.10: *Virtual Sense Hat in the browser*

8. Modify the values as you wish and look at the `Debug messages` tab in the right pane of the NodeRED UI. You can bring such tab to the foreground by pressing the debug icon marked with a red circumference in the following image. NodeRED delivers messages whose format is JSON as you can see in the following view of the `debug` tab:

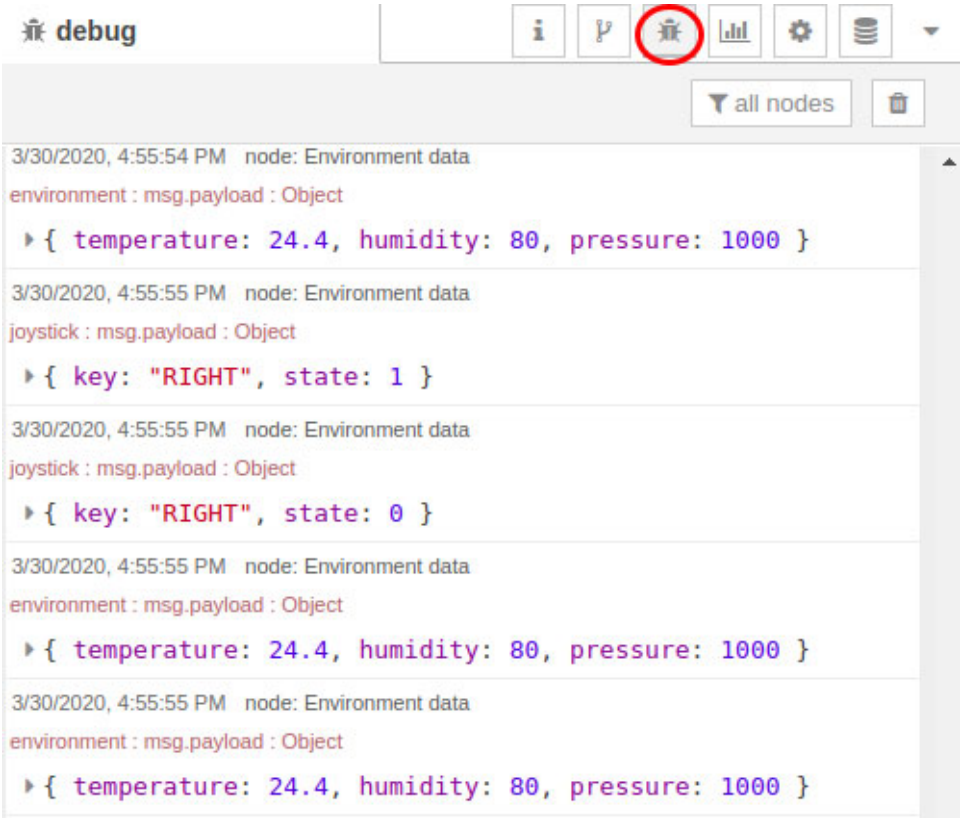


Figure 3.11: *Debugging messages of the virtual Sense Hat*

9. Let's inspect the output of messages from the NodeRED flow, whose structure was briefly explained in the chapter before, in the subsection *Creating your first flow*. There is one type of message in the feed that reports the environmental conditions twice per second, and looks like the following sample:

```
{
 "topic": "environment",
 "payload": {
  "temperature": 24.4,
  "humidity": 80,
  "pressure": 1000
 },
 "_msgid": "76a8fa54.88b774"
}
```

The first field—`topic`—tells the type of message, i.e., `environment` in this case.

The second field—**payload**—carries the values of the 3 variables that stand for the **environment** state: **temperature**, **humidity**, and **pressure**. Whenever you modify some of them in the simulator window (remember step 6 above), the next environment message will immediately reflect the updated status.

The third field—**msgid**—is a unique identifier of the message to be used internally by NodeRED.

10. The other type of message corresponds to joystick events like the following one:

```
{
 "topic": "joystick",
 "payload": {
  "key": "RIGHT",
  "state": 2
 },
 "_msgid": "fa72d84b.ec9e78"
}
```

In this case, the **topic** field tells that the message corresponds to a **joystick** event. The payload contains two values:

- The **key** attribute refers to the direction in which the joystick is moved: UP if you pressed the up arrow key in the simulator window, DOWN if it was a down arrow keystroke, LEFT for the left arrow key, and RIGHT if you pressed the right arrow key.
- The state attribute may take one of the values:
  - 1 for a single press
  - 2 for a long press
  - 0 when the key is released

Hence, a single press sequence will be 1-0, while a long press sequence

will be 1-2-0.

In this section, we have installed a new package, `node-red-node-pi-sense-hat-simulator`, and in the *Installation in the host OS* section of the previous chapter, when describing the dependencies of NodeRED, we mentioned that we would later explain the meaning of the version numbering of npm packages. And that is what we are going to do next.

Semantic versioning in Node packages

To inspect the installed packages as well as their versions, type in the following lines in a bash terminal from the `app` folder of the repository:

```
$ cd ~/book_hands-on-iot/chapter3/app
$ npm list --depth 0 | grep red
```

We are asking to list all npm packages installed in the current folder (`npm list`) showing only the first dependency level (`--depth 0`), and filtering the packages that contain in their name the string "`red`" (piping command in Linux is accomplished by the operator "|"). The output of the command should be similar to this:

```
hands-on-red-iot@1.0.0
node-red@1.0.5
node-red-admin@0.1.5
node-red-contrib-device-stats@1.1.2
node-red-dashboard@2.19.4
```

`npm` packages are versioned using three numeric codes separated by dots. Let's take the example of `node-red` itself, whose current version number is 1.0.5:

- The first value, 1, is the major release number. This level is applied when there are new relevant functionalities and major changes in the code. Major releases may include breaking changes in the package, not ensuring backward compatibility with the previous major release.

Tip: For every other package depending on node-red, we should not update the version in its package.json when the release 2.0.0 of node-red is published. If it were done, such a package could give up

working since backward compatibility with node-red 1 is not assured. We should wait until the package owner releases a major version compatible with node-red 2.

- The second value, 0, is the *minor release*. This version can be safely updated in your app since it will not include breaking changes in the code.
- The third value, 5, is the *patch release*, and is used to solve bugs.

To finish this explanation, we should be aware of how to specify dependencies in the file **`package.json`**. Let's have a look at this section for **`node-red`**:

```
"dependencies": {
  "node-red": "~1.0.4",
  "node-red-admin": "~0.1.5",
  "node-red-dashboard": "~2.19.4”,
  "node-red-contrib-device-stats": "~1.1.2"
}
```

The symbol "~" prepended to the version number of every package means that, when running **`npm install`**, it will install the latest patch release of every package. If instead you put the symbol "^", it will install the latest minor release, which is also safe because it does not add breaking changes in the code.

At this point we will introduce a very useful feature to keep our work organized. It consists of the NodeRED projects feature, a set of flows that constitute a single application, like the digital clock of the first chapter.

NodeRED projects

Each project, apart from being a NodeRED entity, will own a separate Git repository so that we can keep track of it, and easily switch to another project depending on the work we need to do in NodeRED.

Let's learn by example creating a new project and configuring its common characteristics. To do so, deploy the right-side menu of

NodeRED, select `Projects`, then `New`, as depicted in the next screenshot:

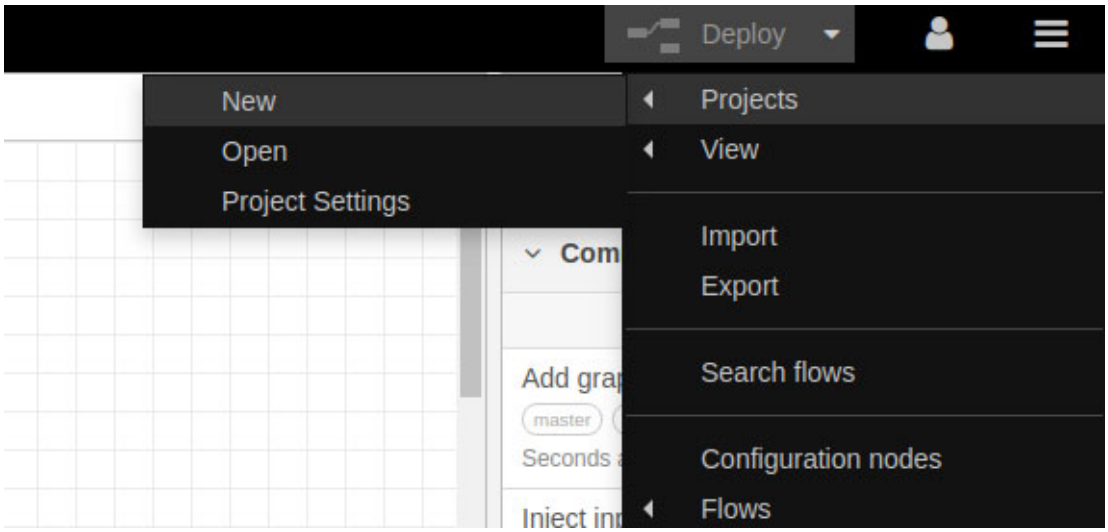


Figure 3.12: *Creating a new project*

Of the two options in the pop-up window, select `Clone Repository`:

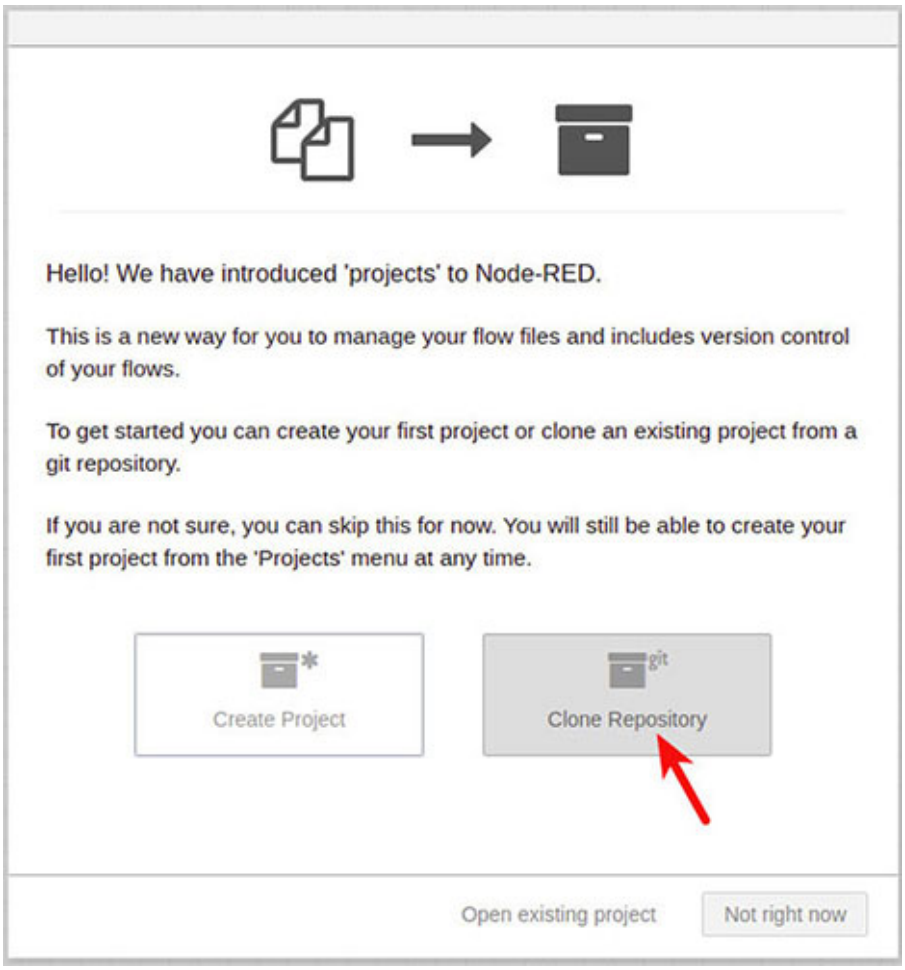


Figure 3.13: *New project from an existing repository*

Since this is the first time you access this feature, you will be prompted to identify yourself by providing a username and an email address for Git management:

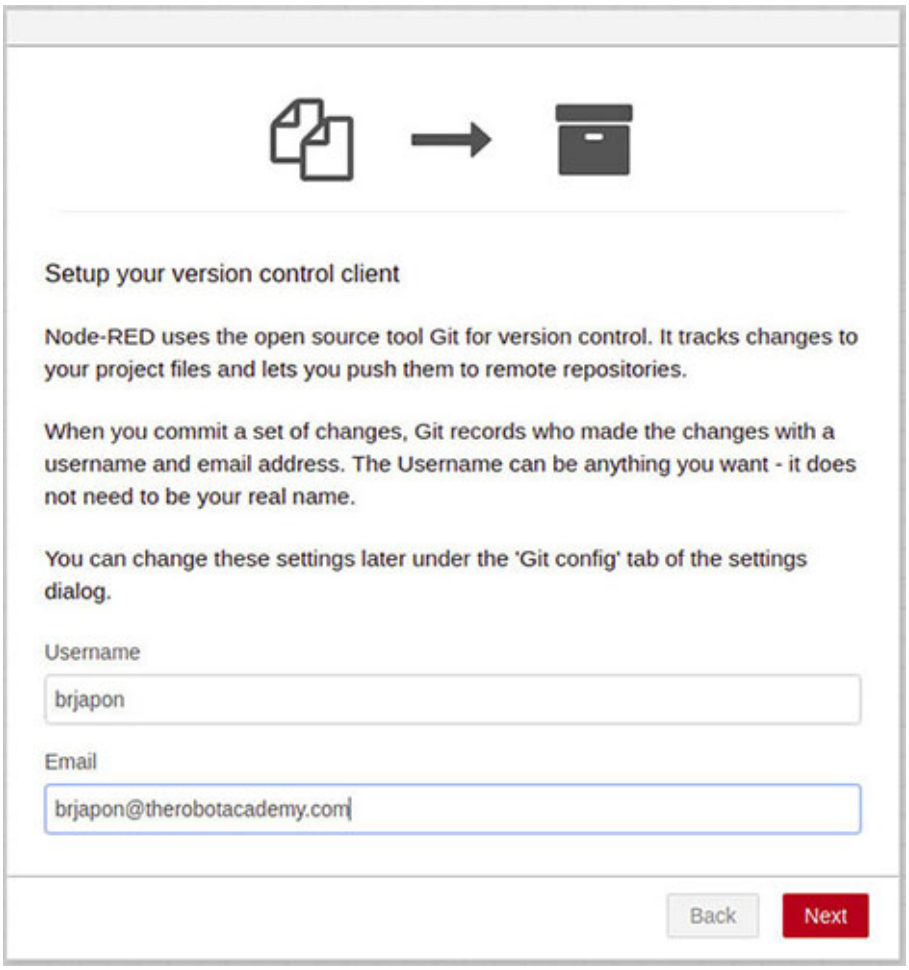


Figure 3.14: *Providing git credentials for the version control of the project*

Note: Be aware that the identification is not required by NodeRED itself, but by git. Since this is a control version tool, it needs to report not only what changes are made to the code in the repository, but also who has been the author. It is quite common that several people, i.e. a team, work in the same repository for the development of an application. Hence, it is essential for the version control to track changes as well as their authoring.

After filling in and pressing the `Next` button, you will be presented with a window where to introduce the information of the repository to clone. We only need to specify the public GitHub repository of the project to clone, i.e. **https://github.com/Hands-on-IoT-Programming/chapter3-sense-hat-simulator**, as shown in the next screenshot:

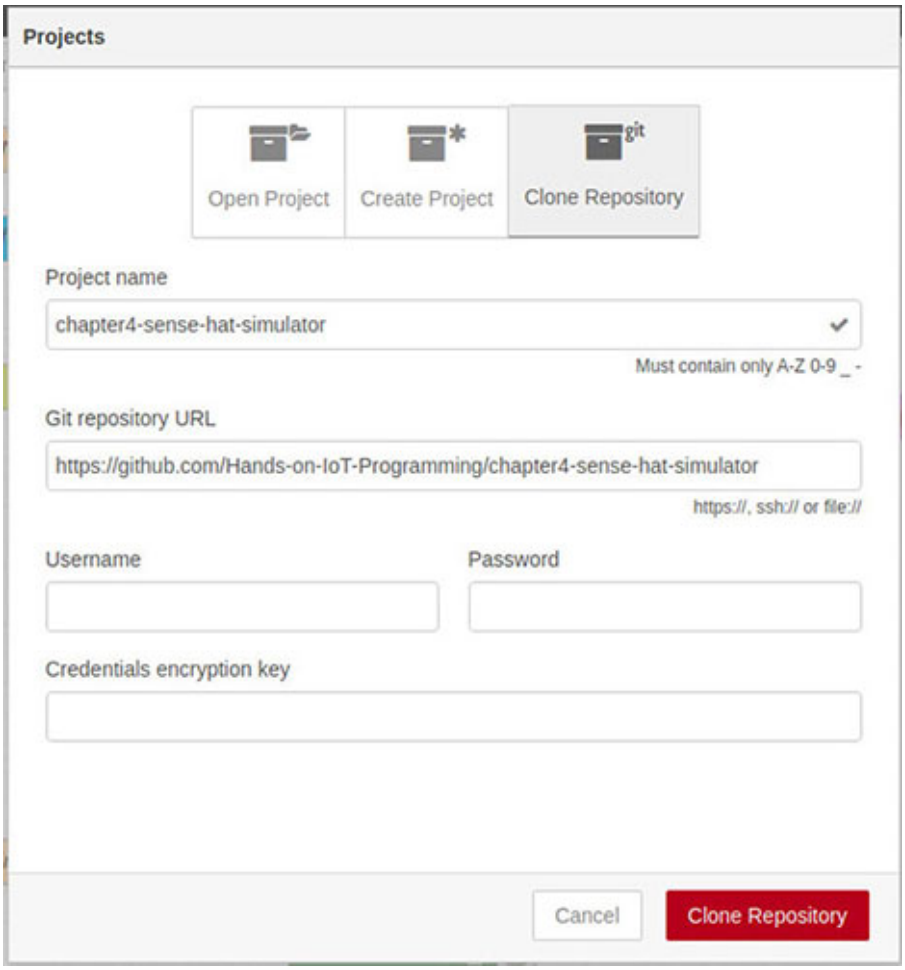


Figure 3.15: *Provide data for cloning an existing repository*

The field `Project name` is autocompleted by copying the repository name. You can give any other name if you want. Regarding the rest of the fields:

- `Username` and `Password` are the credentials to access the remote Git repository that you are cloning. Since all the codes of the book is hosted as public GitHub repositories, you can leave blank both fields.
- `Credentials encryption key` is a safety feature added by NodeRED to the project, because you have the choice of saving your project encrypted so that only people with the right key may access its contents. Leave this field blank since the sample project is not encrypted.

Once cloned, you will see in the workspace the same flow that we built in the previous subsection *Setting up a virtual Sense Hat in NodeRED*. Hence, you may review steps 6 to 10 of such subsection to check that it behaves the same.

Tip: Be aware that we are showing the solution to the practical exercise in the previous subsection Setting up a virtual Sense Hat in NodeRED by loading a premade NodeRED project. Remember that in the previous chapter, we showed the solution to the digital clock project from another flow file that was loaded by launching NodeRED with the command npm run solution. Recap the subsection. Review the solution to review the details.

After loading the project, it seems that you have lost the previous work you did on your own before. Then, what if you want to recover the work you did outside the current project? In the following section, we show you a simple hack to restore it.

Exporting flows

It happened that, as our first project in NodeRED is cloned from a Git repository, it disappeared in the flow you were building in the subsection *Setting up a virtual Sense Hat in NodeRED* above. There is a trick to recover it, just follow the next steps:

1. Shutdown NodeRED, i.e., stop or close the terminal from where you launched the app.
2. Open with a text editor, the file **`settings.js`** located at the folder of the NodeRED code, i.e. **`~/book_hands-on-iot/chapter3/app/.`**
3. Look for the projects option at the end of the file:

```
projects: {
  enabled: true
}
```

4. Change the enabled option to false.
5. Start NodeRED again, go to the browser, and find that the initial flows are back.

Note: Be aware that you cannot open any project now, since you have disabled that option in the settings file. We will restore the initial configuration once we have exported the initial flows to a safe location.

6. Select the **`Export`** option of the NodeRED editor menu.

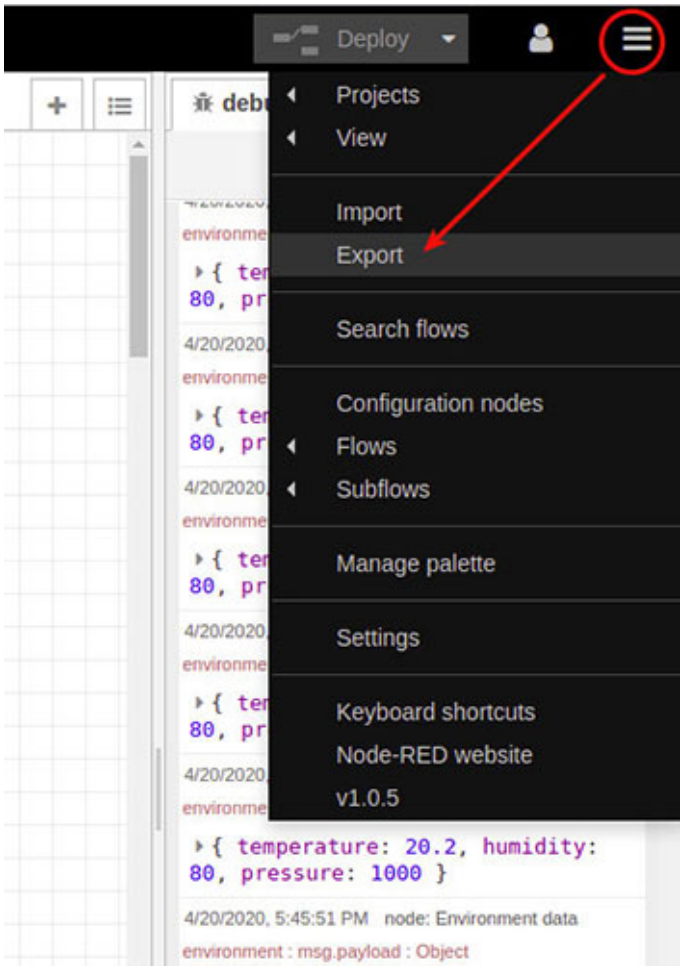


Figure 3.16: *Exporting NodeRED flows*

7. A window will pop up letting you select what you want to export. Select **`all flows`**.

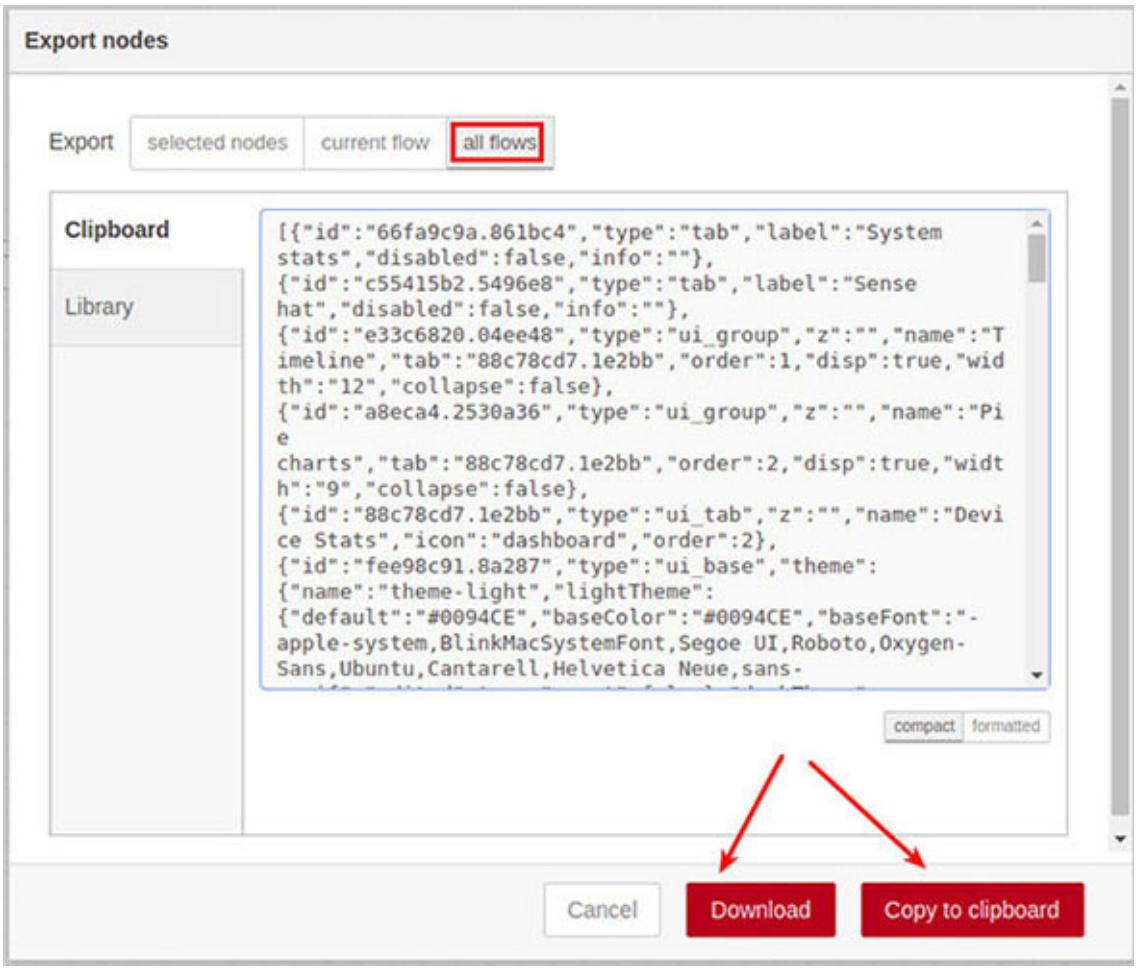


Figure 3.17: *Select what is to be exported*

8. Then, you have two options to save them, as shown in the figure above: download and get the JSON file, or copy the whole, export to the clipboard and move it wherever you want.
9. Re-edit `settings.js` to switch back the projects option to true.
10. Start NodeRED again, and your project enabled application is back with the last project you opened in the foreground.

To finish this subsection, let us see how to manage the specific dependencies of NodeRED projects.

Install project dependencies

To access this feature, select the `Project Settings` item from the NodeRED editor menu, as shown in the next figure:

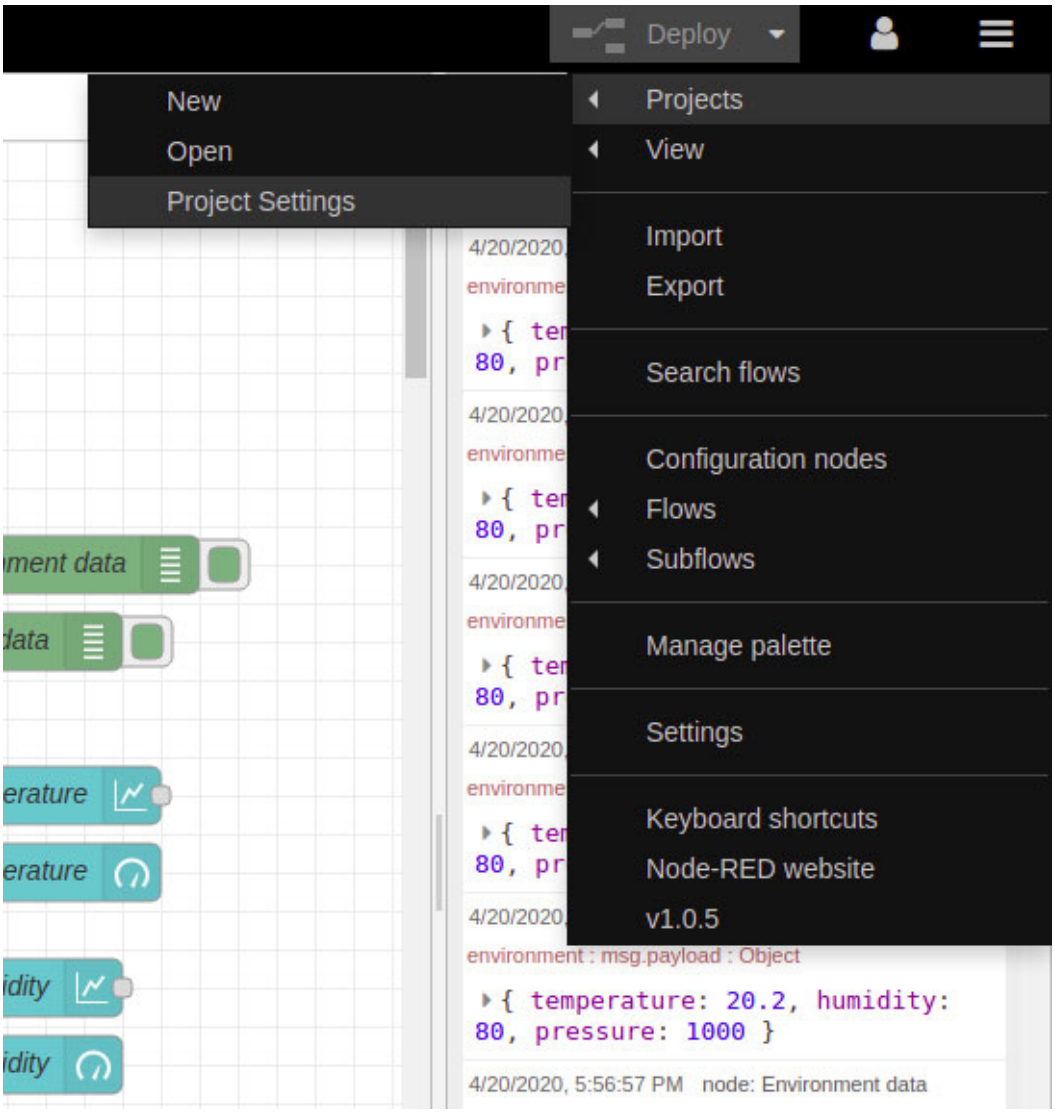


Figure 3.18: *Opening project settings*

A window will pop-up, and in the middle tab there are the dependencies item as shown in the next screenshot:

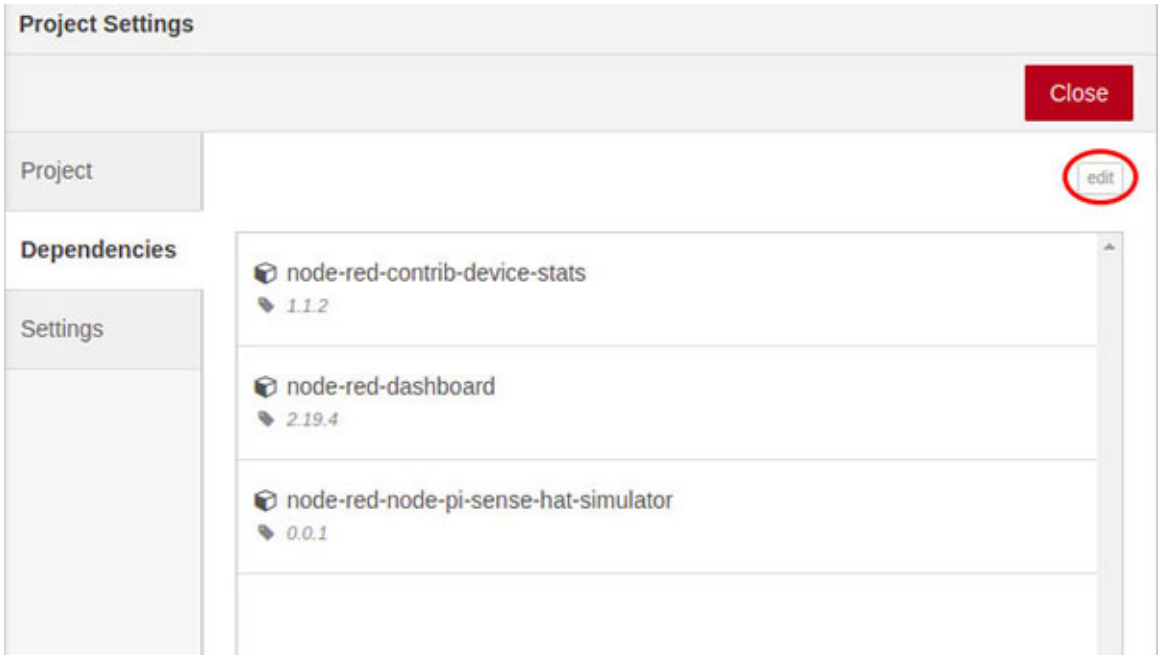


Figure 3.19: *Accessing the dependencies tab for installing or editing*

If any of the dependencies were missed, an install button would appear to the right of the package name, and by pressing it the issue will be solved.

You also can get access to edit by hand the dependencies section of `package.json` file, by just pressing the edit button in the top right part, (see the red circle in the image above).

Once we have correctly set up the NodeRED project, in the last subsection, we will review the dashboard, part of the acquisition functionality that lets us see the time series of environmental conditions.

Data acquisition

The goal in this subsection is to keep track of the last 100 measurements of every environmental variable: temperature, humidity, and pressure. In the project we have loaded, there is also the flow of system stats with which we started the description of NodeRED in the previous chapter. The following image shows the complete flow:

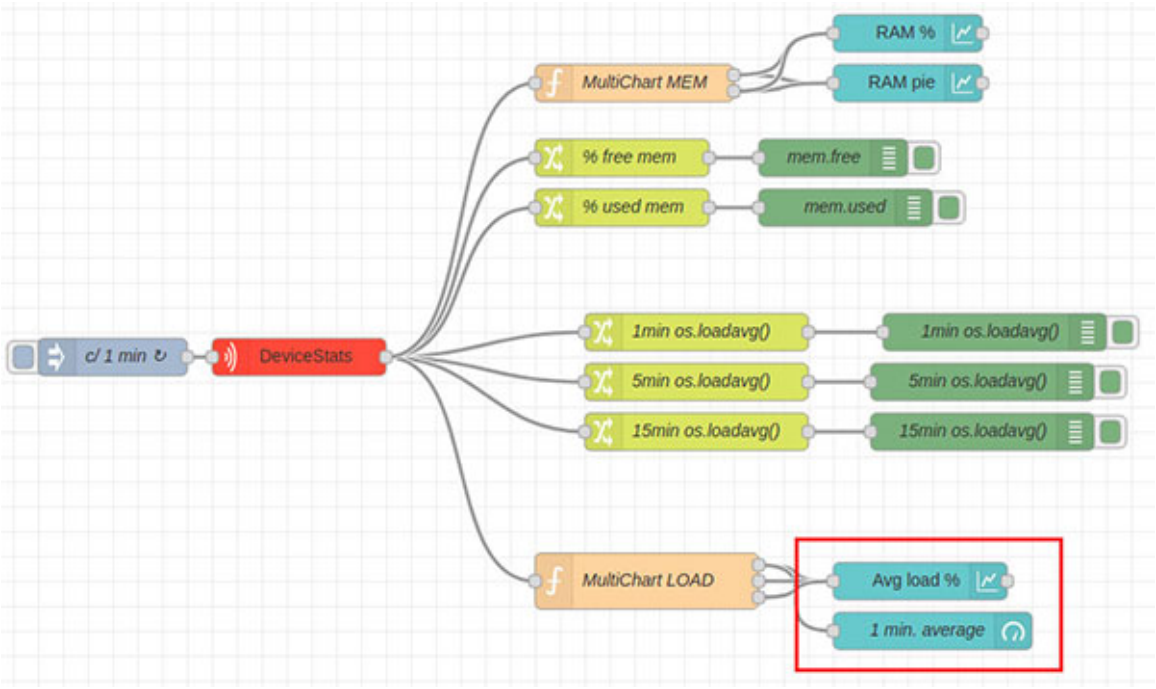


Figure 3.20: *Nodes for the dashboard of the system stats flow*

The two nodes inside the rectangle are the ones we need to use: the upper one graphs the historical series and the other- the gauge- provides the current value of the variable. Remember that these nodes can be found in the `dashboard` category of the `nodes palette`. Find them (chart and gauge) enclosed in a rectangle in the following screenshot:

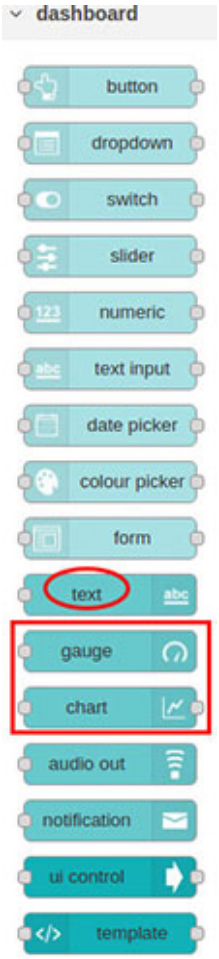


Figure 3.21: *Nodes of the Dashboard package*

The node enclosed inside a circle, text, is the one we used for the dashboard of the digital clock example in *Chapter 2, Getting Started with NodeRED*.

Creating graphs and gauges

In order to build the dashboard, you could just copy and paste `chart` and `gauge` nodes from the `System stats` flow to the `Sense Hat` flow. Following the steps detailed in the Building the clock widget section of the previous chapter, you can easily build the dashboard for the three variables of interest. The result should be similar to the following image:

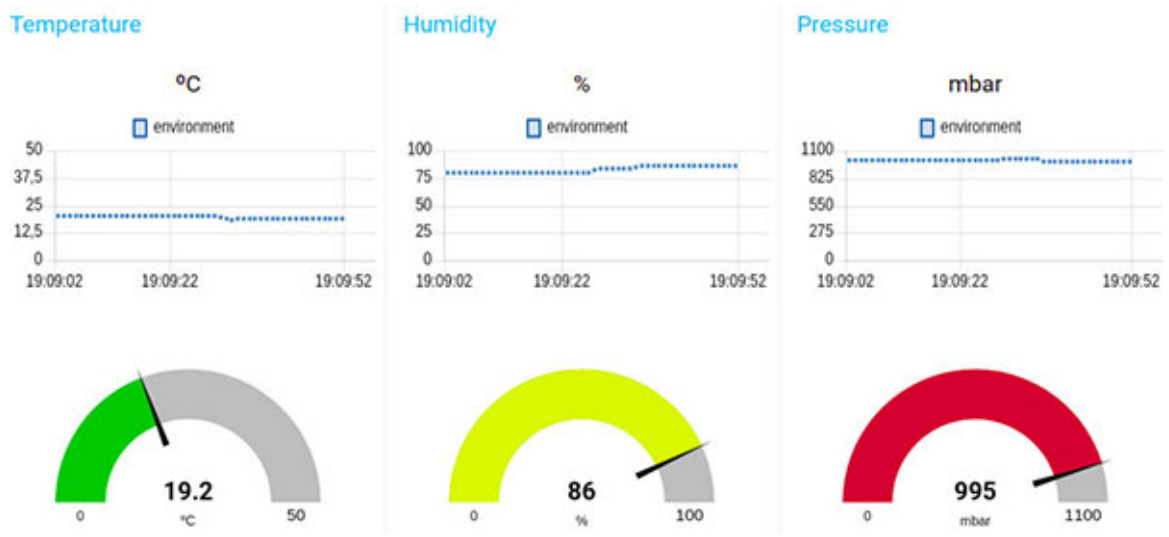


Figure 3.22: *Dashboard UI for the environment conditions*

Since we want to keep track of the *last 100 measurements*, we only have to configure it for the three graph nodes. Let's take the example of temperature. By double-clicking on its graph node, the following form should appear:

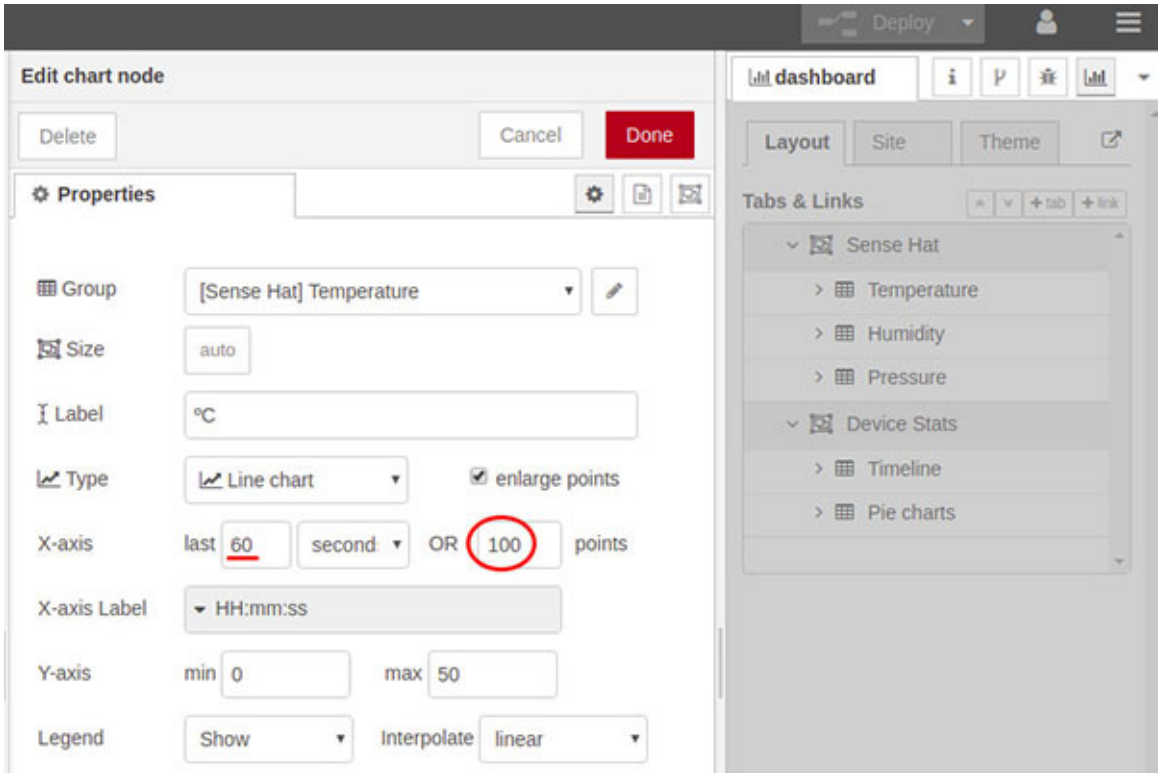


Figure 3.23: *Configuration of the Chart node*

You have to specify 100 points in the X-axis field. Be aware that there is an OR condition with respect to time, i.e., 60 seconds. The graph will show the shortest span between both. Since we have a measurement every 0.5 seconds, in 60 seconds we obtain 120 points. So, the widget is correctly configured to provide at least the last 100 measurements.

The rest of the chapter is going to be devoted into teaching how you can broadcast the environmental data to the outside world, so that consumers may receive data and perform analysis for the purpose of their specific applications.

Real-time streaming apps with Pusher

At this point, we have a continuous data stream from Sense Hat ready to broadcast to the Internet so that we can perform the data processing on any remote server, i.e., the consumers. This operation of serving the real-time stream of environmental conditions to the consumers is what constitutes the *streaming* functionality.

There are several commercial options as well as Open Source solutions to implement this function. We have chosen Pusher as a **Software as a Service** (**SaaS**) solution that lets you quickly perform the integration without needing to do any server deployment on your side.

SaaS solutions are a cost effective way to add services to your app without the pain of deploying your own servers and skipping any maintenance effort. You are a user of the service, and pay only for the actual resources you use:

- For *companies,* this means to act as consumers of the services, skipping the need to devote skilled IT personnel to the maintenance. This is provided by the vendor, who offers a *Service Level Agreement* that guarantees the quality of the service in terms of availability, scalability, and technical support.
- For *individual developers,* it means a quick way to integrate the service in the infrastructure of their apps because vendors usually offer a free developer plan (or sandbox server free of charge). This allows for rapid prototyping of the app with minimum cost.

On the other side, Open Source solutions are a frequent choice in companies that need to host the infrastructure behind their firewall for security reasons. Also, for individual developers it is the way to learn and master the technology behind (real-time streaming in this case) to decide later if going to a SaaS solution or maintain their own server.

Tip: As an Open Source solution, there is a good choice that is Deepstream https://deepstream.io/ . It consists of a professional solution for data streaming written in Node, the same environment as NodeRED. It is easy to set up and run, and also provides a layer of persistence for data synchronization (state data of the app is stored in a database integrated with Deepstream out of the box).

Since the orientation of this book is at *system level* and we have to integrate heterogeneous services, as a general approach we have the rule to opt for SaaS solutions free of charge for developers. In the case of the streaming functionality, this is the reason to decide for Pusher. Hence, this section will be devoted to guide you in order to set up this service and connect it to the data stream from NodeRED.

Set up your Pusher account

Follow the next steps to create and configure your free Pusher account:

1. Visit **https://dashboard.pusher.com/accounts/sign_in** .You may sign in with identity providers such as Google or GitHub, or you can set up an account from scratch. Take the choice you prefer.

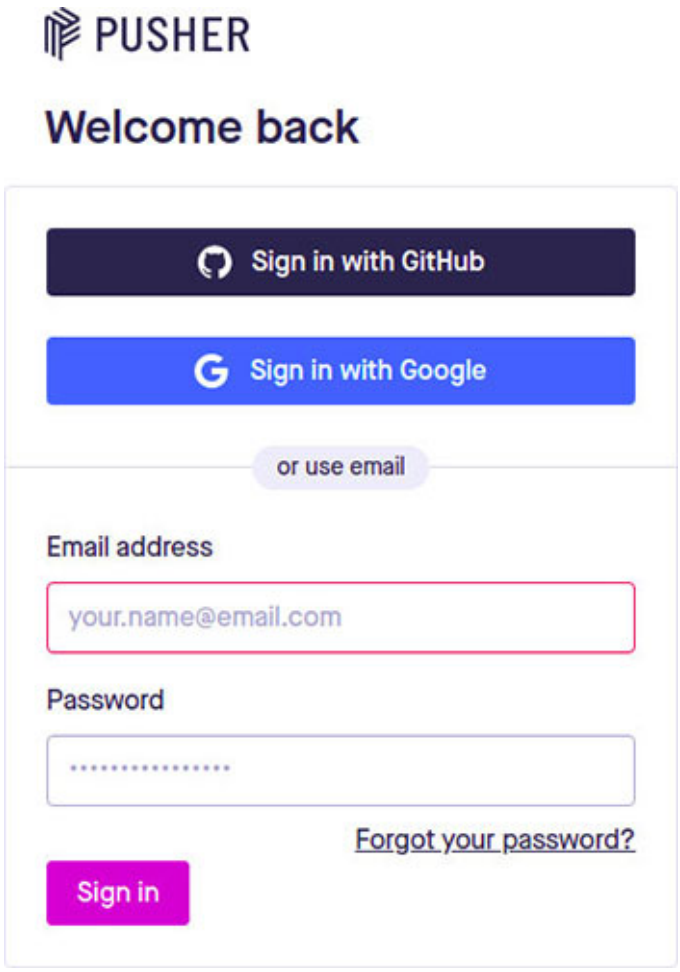


Figure 3.24: *Sign-in window of Pusher*

2. In our case, we will sign in using a GitHub account. Hence, select this option and authorize the access as shown in the next image.

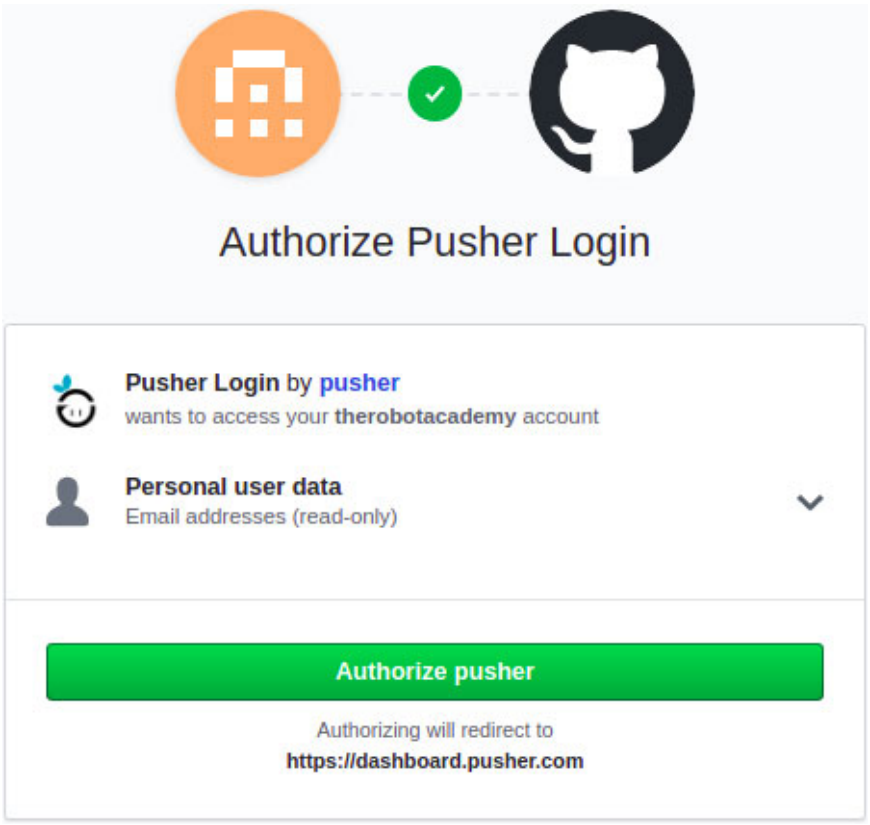


Figure 3.25: *Authorization of your identity for using Pusher*

3. The first time you login, the application guides you in order to make it easier to get started. Hence, you are presented a pop-up window where you can create your first app, which consists of a Pusher channel where you will feed environmental data. Name the app as `environment`, and select the server nearest to your location:

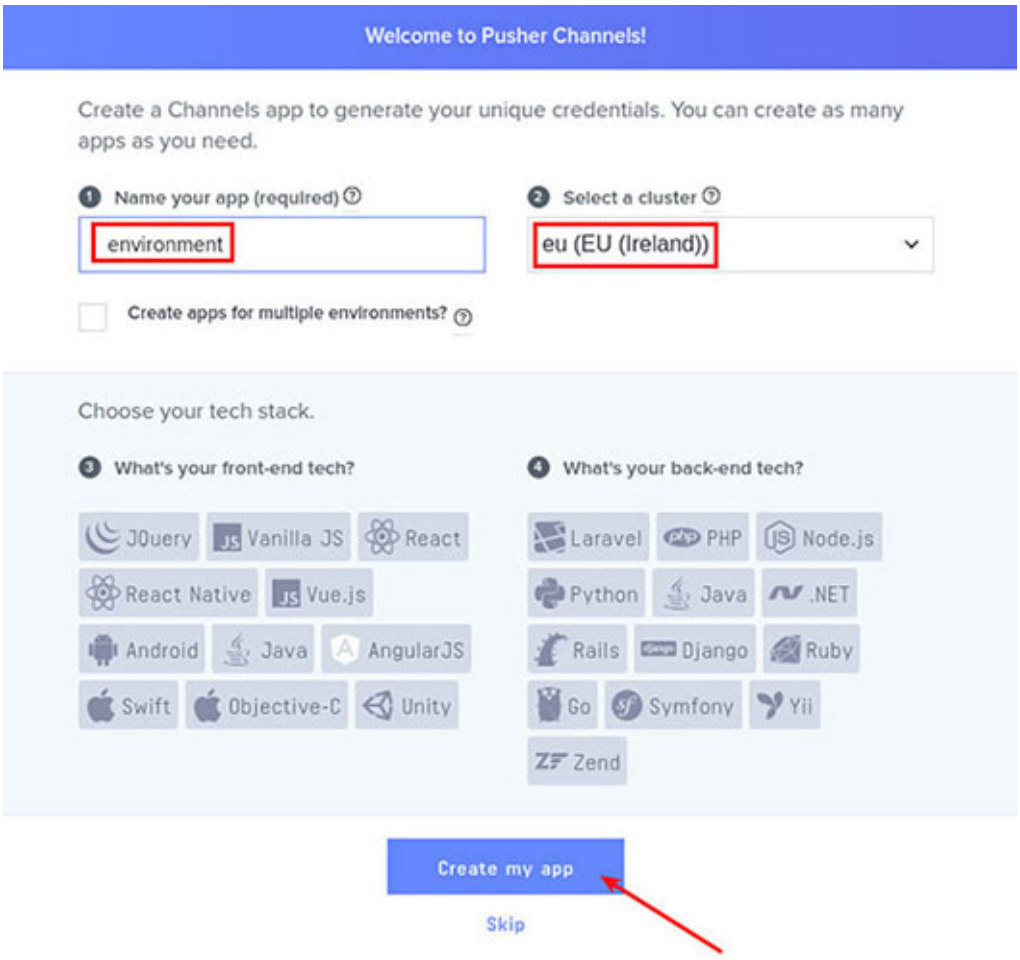


Figure 3.26: *Creating the first app in Pusher*

Finally, press the button **`Create my app`**, and then, you will be forwarded to get the configuration data of the channel.

4. You are driven to the **`Getting Started`** item of the dashboard menu, enclosed in a rectangle in the image below. Here, you have code snippets to integrate the Pusher channel with your client and server. Since NodeRED provides a contributed Pusher package, you do not need to code anything, just configure its NodeRED Node as we will show in the next subsection. Click on the **`App Keys`** item of the menu marked with an arrow in the image.

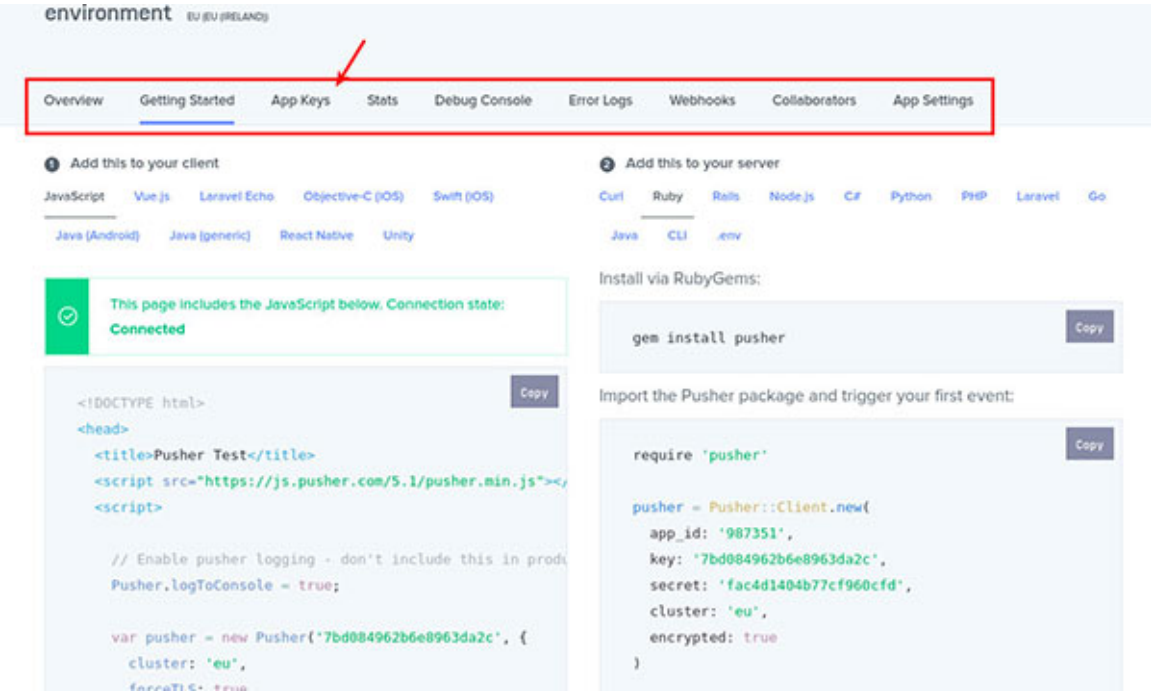


Figure 3.27: *Getting started with Pusher*

5. In the `App Keys` tab, you are presented with the key and secret that you will have to include in the NodeRED application in order to connect the channel. Take note of the credentials.

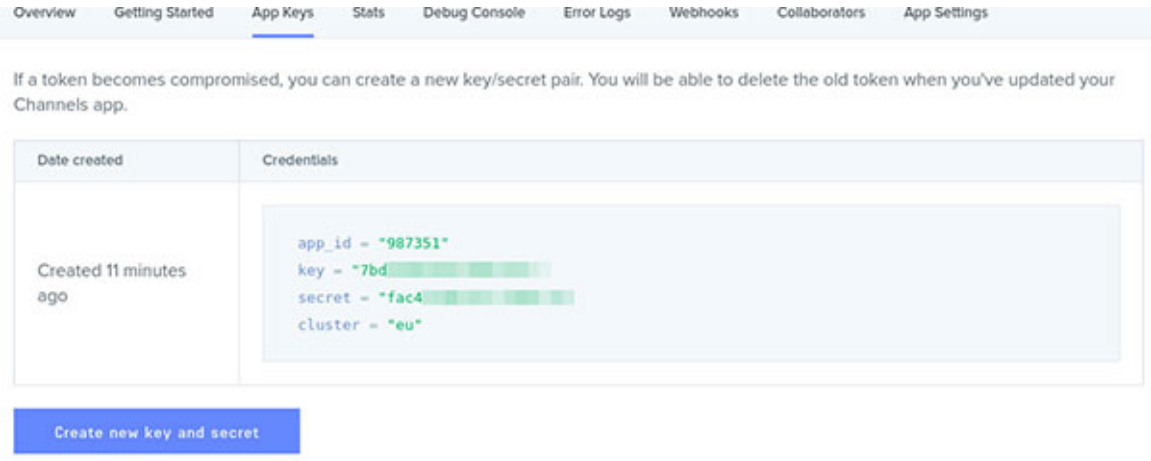


Figure 3.28: *App keys for integrating Pusher into NodeRED or any other app*

At this point, we have finished the configuration in the Pusher dashboard. The next step is to set up the Pusher client in NodeRED.

Connecting NodeRED to Pusher

This step is pretty straightforward since what you will do is basically what you have to do every time you install a new NodeRED package:

1. Install the module from the `Manage Palette`.
2. Add a node of the new package to the workspace.
3. Double-click on it and fill in the form to configure the Node.
4. Connect it to your flow.

The NodeRED package for integrating Pusher has two nodes, pusher in and pusher out. You can check the details at the URL of the module **https://flows.nodered.org/node/node-red-node-pusher** . Find it in the Manage Palette and install it as shown in the next image:

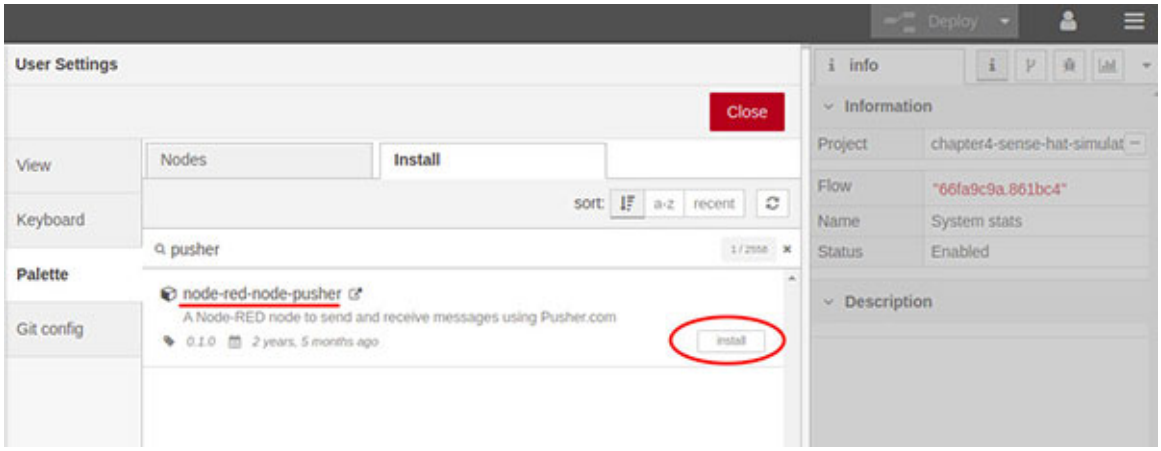


Figure 3.29: *Installing node-red-node-pusher package for Pusher integration*

You can access the new nodes under the category mobile of the nodes palette:

Figure 3.30: *Nodes of the node-red-node-pusher package*

Since in this chapter we want to publish data, you will now use the pusher out Node. In *Chapter 4, Real-time data processing with NodeRED*, we will use the other node, `pusher in`, to subscribe to the

environment channel and obtain real-time updates on which to perform analyses.

Publishing environmental data from the Sense Hat

Add two nodes to the flow you built for the Sense Hat in the previous section:

- one `debug` node to check the data that are being published
- one `pusher out` node to broadcast such data

They are enclosed inside a rectangle in the following image:

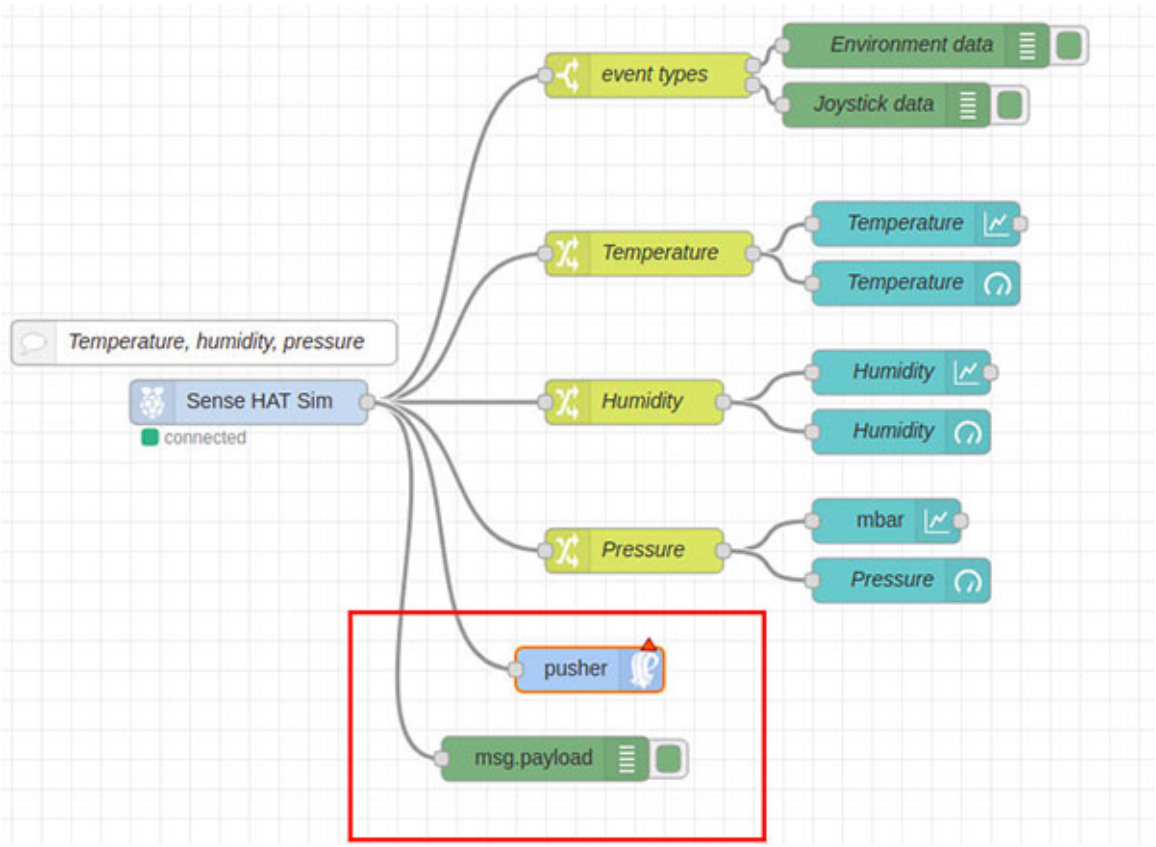


Figure 3.31: *Node for broadcasting Sense Hat data to Pusher*

Note: When you deploy the flow with the new nodes, you will receive a warning due to the fact that pusher out Node has not been configured yet. Deploy anyway, because at this point, you need to

check if the environmental data is being published properly by checking the output of the debug Node.

Once deployed, check in the **`Debug messages`** tab of the Side bar that **`msg.payload`** contains the three environment variables to track:

```
{
  "temperature": 20,
  "humidity": 80,
  "pressure": 1000
}
```

Then, open the **`pusher out`** Node and configure it with the credentials you obtained from the Pusher dashboard. The following screenshot shows the correspondence so that you can proceed straightforwardly:

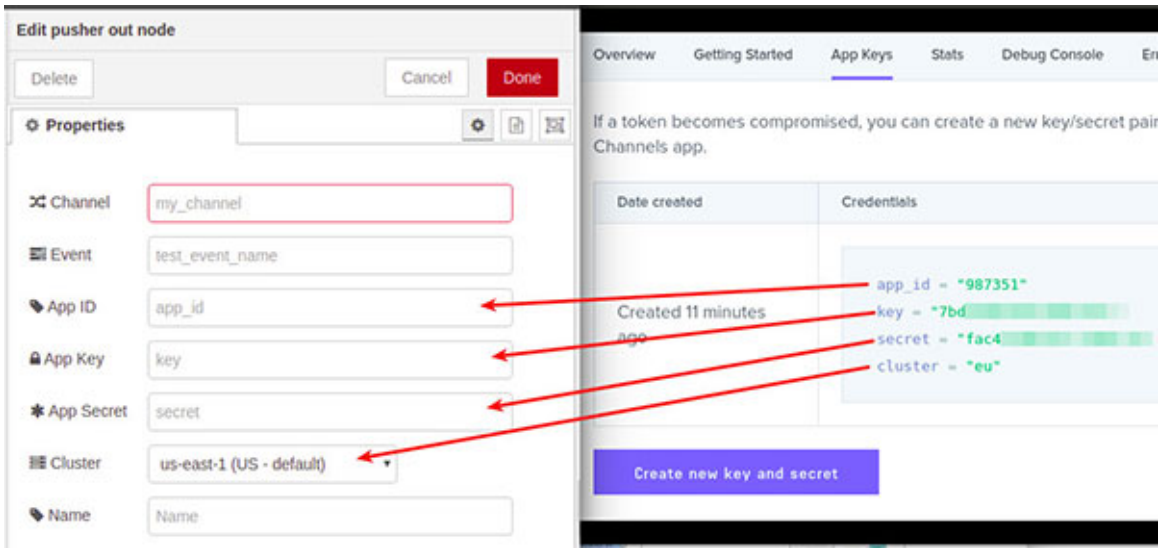


Figure 3.32: *Configuring the Pusher out node*

`Channel` and **`Event`** fields are internal for your NodeRED flow and you can name them as you like. Both are the attributes that will be included as fields in the **`msg`** object. In particular, the Event attribute will be the **`topic`** field of every message.

Once deployed, if you do not receive any warning or error, data should be published to Pusher. We will do the check by adding a new tab to the workspace as follows:

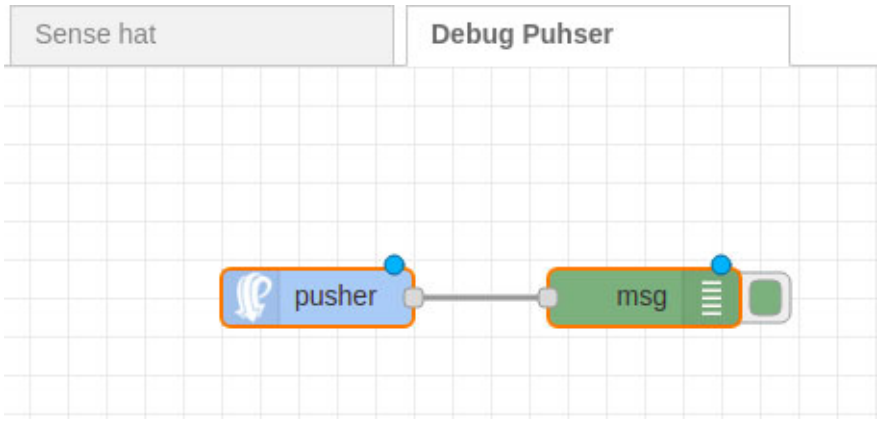


Figure 3.33: *Receiving data from the Pusher channel*

Looking at the code of the chapter, you will find this flow in the tab called `Debug Pusher`. In this case, we are adding the Pusher in Node, and we should configure it as follows to make sure it connects to the right Pusher channel:

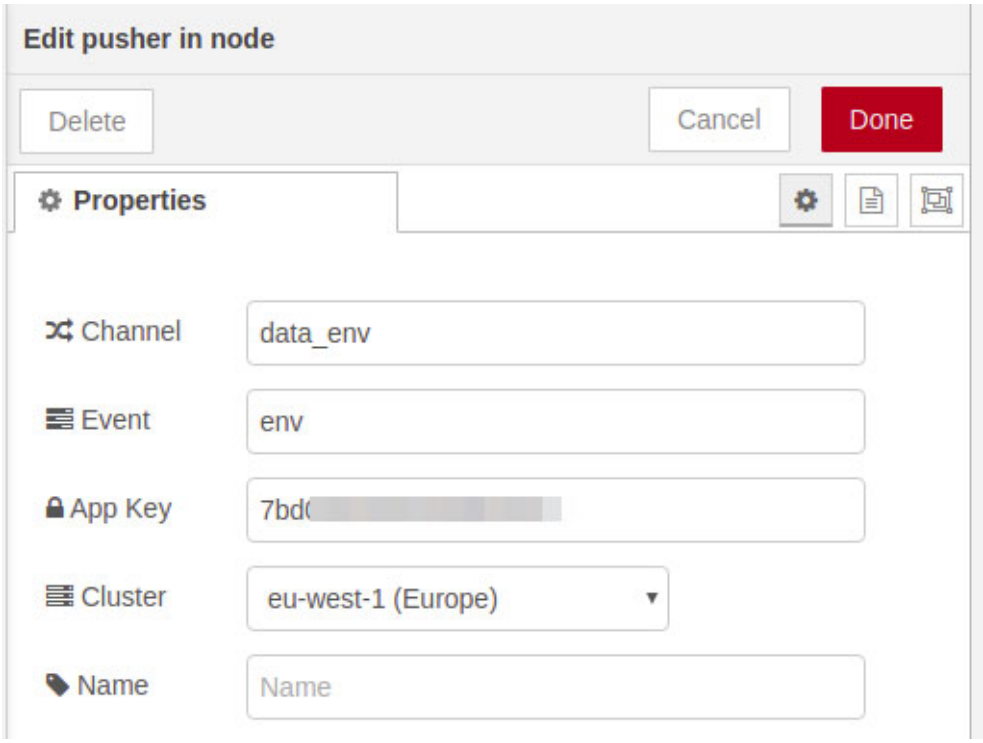


Figure 3.34: *Configuring the Pusher in Node*

The only specific fields you need from your Pusher app are the `App key` and `Cluster`. The `Channel` and `Event` should coincide with the names you gave in the `pusher out` Node.

Finally, configure the **`debug`** Node so that it reports complete messages. Deploy the flow and you should see the following output in the **`Debug messages`** tab:

```
{
  "topic": "env",
  "channel": "data_env",
  "payload": {
    "temperature": 20,
    "humidity": 80,
    "pressure": 1000
  },
  "_msgid": "5d2203cf.4c8c5c"
}
```

If it happens so, you will have checked to be receiving real-time updates of the environmental conditions. Finally, check the Pusher dashboard—**`Overview`** tab—to see the graphs reporting the count of the messages over time:

Figure 3.35: *Overview of the Pusher app stats*

You can see that the publishing rate is very high, i.e., 20 messages every 5 seconds. This is what produces the Sense Hat simulator node. Since environmental conditions do not change in the scale of seconds,

but in the order of minutes, we really do not need such a high publishing rate.

We can limit the number of messages published per second using another NodeRED package called `node-red-contrib-throttle`, whose documentation you can find at **https://flows.nodered.org/node/node-red-contrib-throttle**. Install it as usual, and place the `throttle` node just before `pusher` in Node. Then, set the feed rate to 1 message per second:

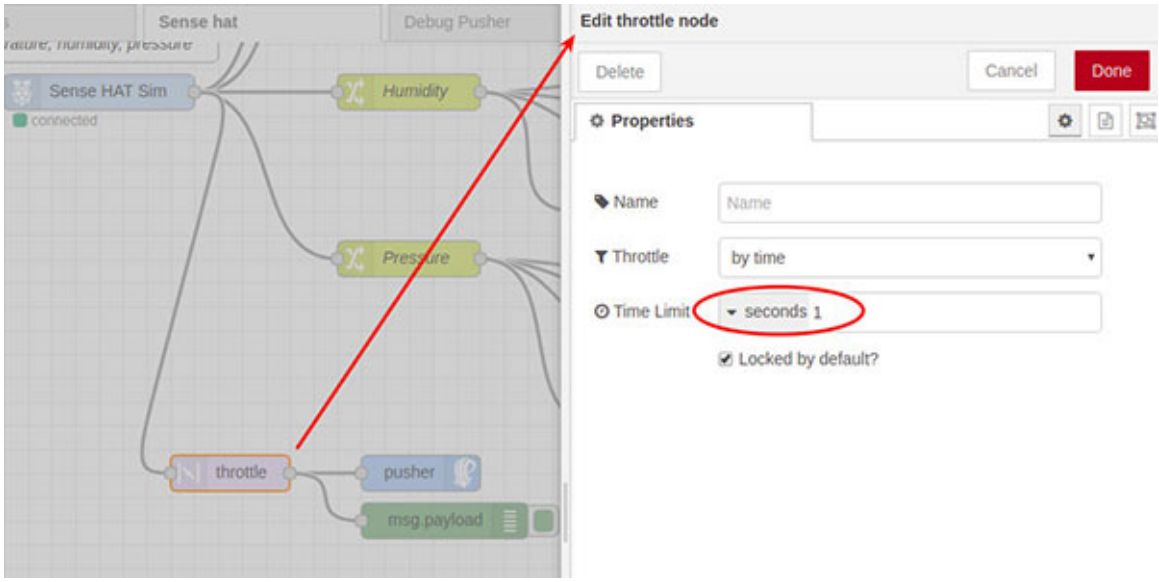


Figure 3.36: *Changing the message release rate of the Throttle node*

Revisit the Pusher dashboard, and check in the live graph how the broadcast rate is dramatically reduced:

Figure 3.37: *Pusher app stats graph showing the reduction of messages per second*

Before processing these data in a remote server (in the next chapter), let's devote a final section to learn how to save the state of variables of your NodeRED flow.

Saving system state in NodeRED variables

There are three possible scopes for the system state in NodeRED:

1. **Node scope**: The variable saved at this level is only visible to the Node from which it was saved. The following two commands allow for saving the state, and retrieving the current value, respectively:
    - `context.set('count', 5)`
    - `context.get('count')`
2. **Flow scope**: The variable saved at this level is only visible to the flow from which it was saved. The corresponding commands are:
    - `flow.set('count', 5)`
    - `flow.get('count')`

3. **Global scope**: The variable saved at this level is visible everywhere in the app, i.e., every flow and every Node. The corresponding commands are:
    - `global.set('count', 5)`
    - `global.get('count')`

The values that you store constitute what we call the system state, since it represents what is happening in the app at every instance. Such a system state is app-specific and has to be defined by the developer when designing the NodeRED flows. For our concrete case, the state is composed of three variables: temperature, humidity, and pressure.

Note: You can set whether the context is stored in memory or in physical files with the option contextStorage in the setting.js file:

```
contextStorage: {
  default: {
    module:"localfilesystem"
},
```

The example above saves the state in physical files, therefore, it persists when you restart NodeRED. The alternative is to set the field module to memory. In such a case, the context is reset if you stop the application since the variables would reside only in RAM memory.

How can we access the system state in NodeRED? Easy, click on the **`Context Data`** tab of the Side bar and you will see the three scopes that we mentioned above:

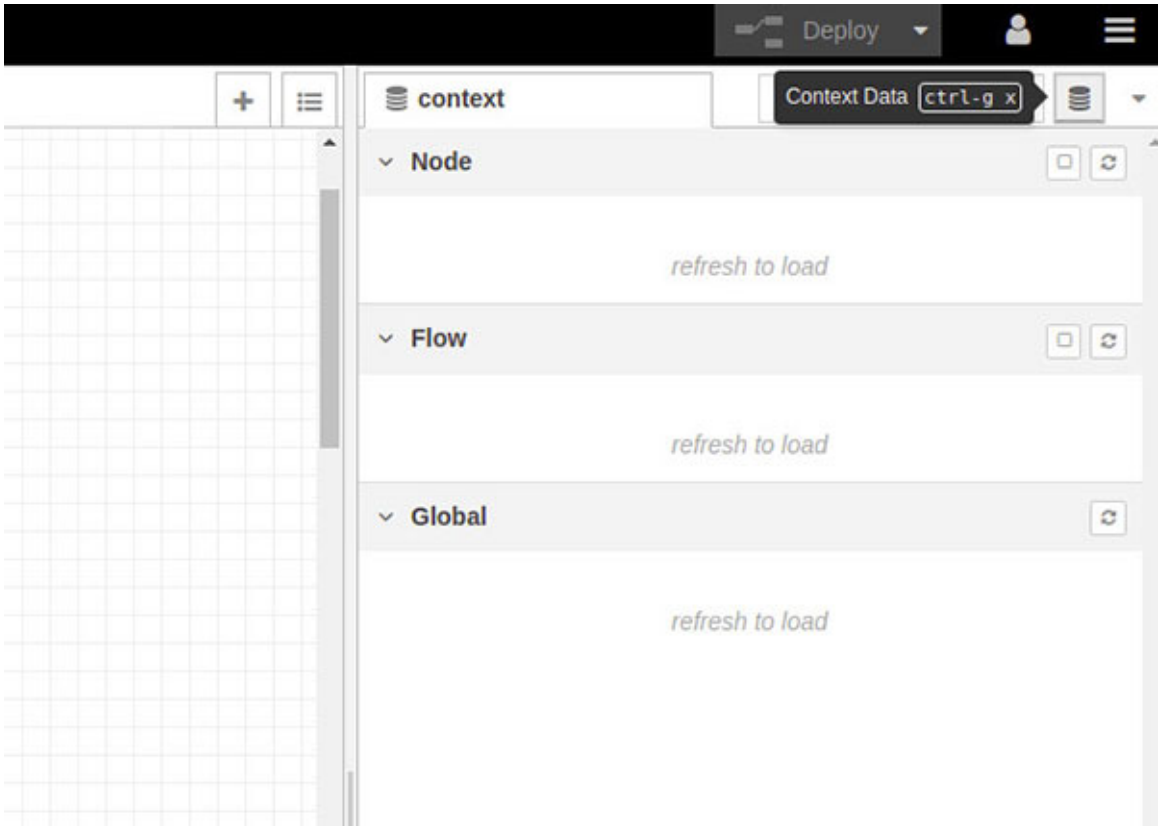


Figure 3.38: *Context Data tab of the Side bar*

At this point, the context is empty, i.e., there is no variable stored. Let's explore how we can persist in the current environmental conditions. It is as simple as modifying the change nodes in the flow as follows:

1. Prepare `msg` objects for each of `Temperature`, `Humidity` and `Pressure`, by opening the existing change node of every variable.
2. For each one, configure `msg.topic` as the variable name, and `msg.payload` as the corresponding value. Follow steps 1, 2 and 3 depicted in the following screenshot:

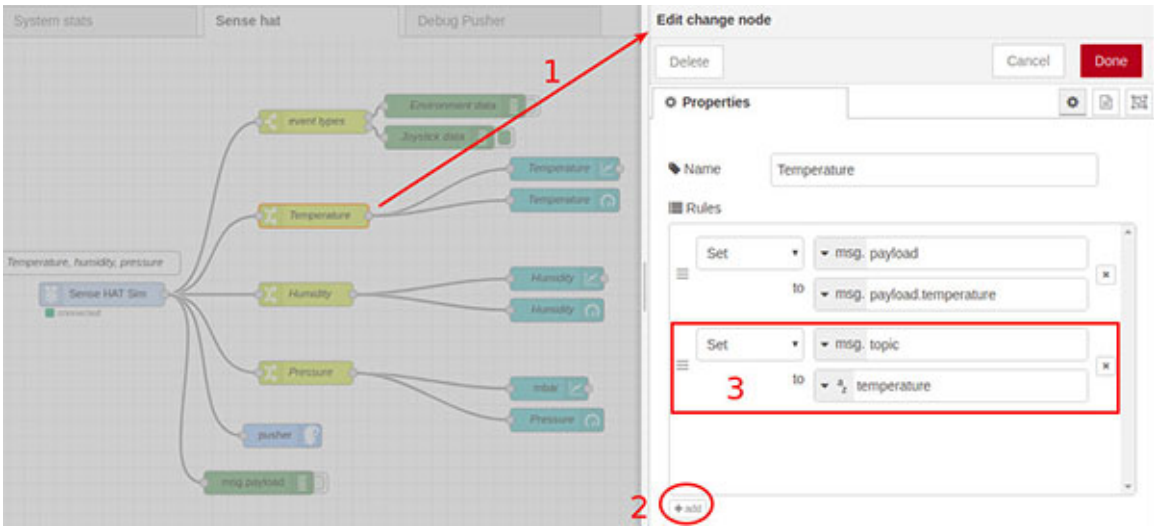


Figure 3.39: *Setting msg.payload to carry only temperature*

3. Add three debug nodes, one for each variable, to check that the output messages are as expected:

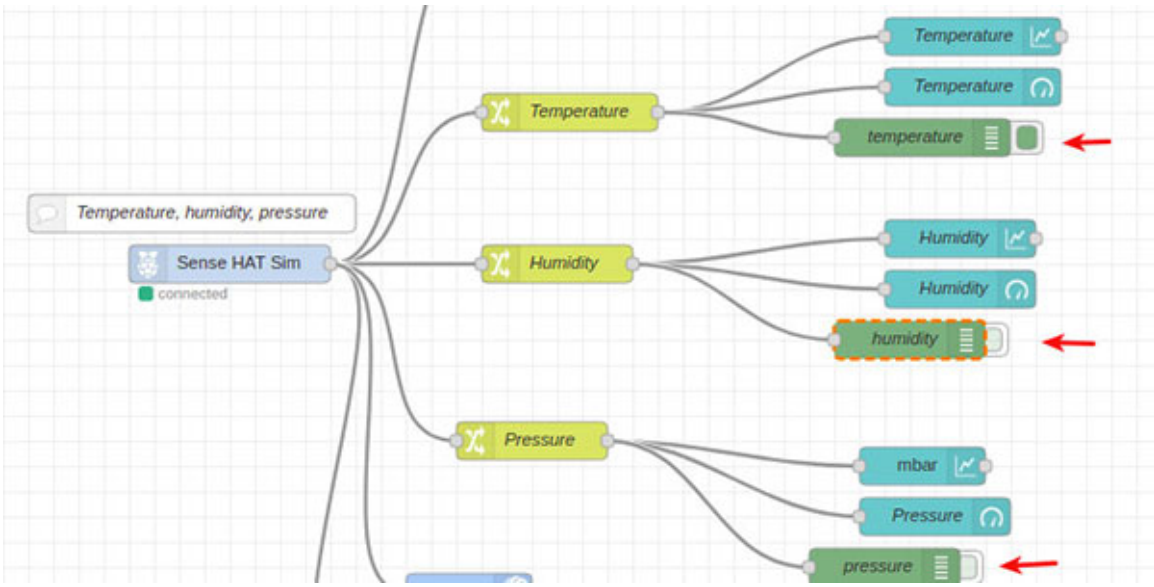


Figure 3.40: *Debug nodes to check the proper configuration of the Change nodes*

4. Add a **`Function`** node from the Node palette, and write the following JavaScript code in its form (see the image in the step 6 below):

```
global.set('temperature', msg.payload)
return msg;
```

The first line saves `msg.payload` of the incoming message into a global variable that is named temperature. The second line ensures that the incoming message is forwarded to the output, and therefore it can be used by other nodes downstream. In this specific case, it is the `temperature` debug Node that we added in step 3 above.

Insert the function node as shown in the next image, between the change and the debug nodes. We gave the name save temperature so that it is self-explicative.

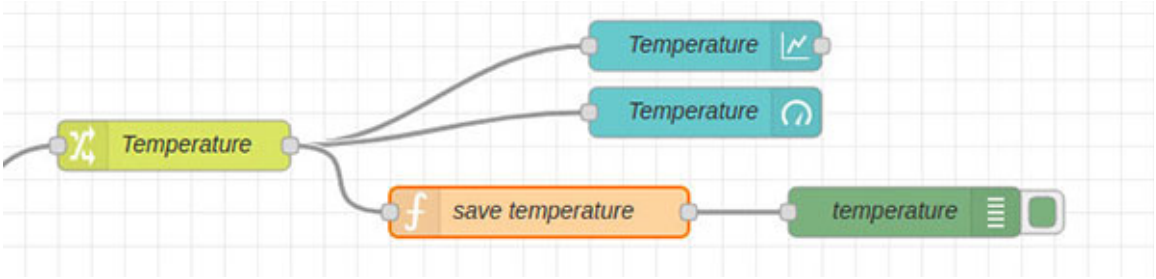


Figure 3.41: *Function node to save temperature in the global context*

5. Deploy the flow and access the `Context Data` tab to confirm that the temperature is saved.

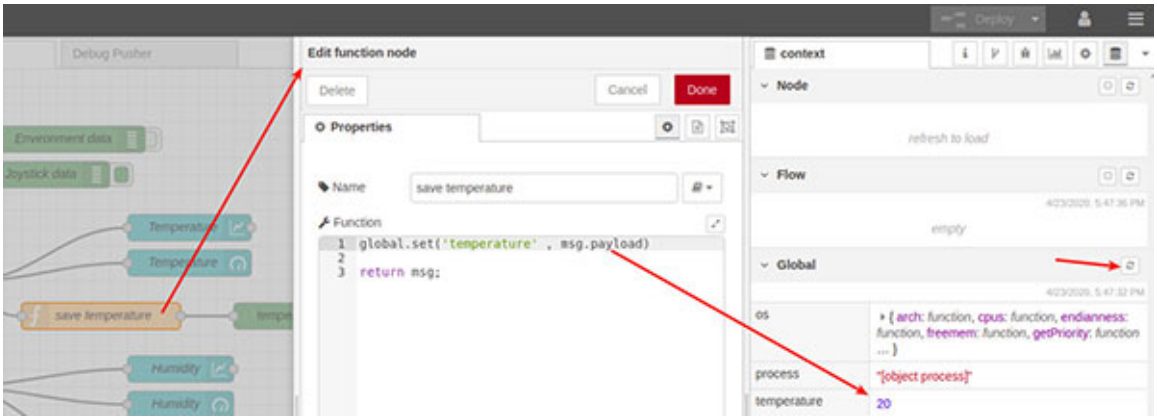


Figure 3.42: *Checking the storage of temperature in the global context*

Be aware that you have to click the refresh icon so that the context window is updated.

6. Finally, in order to access the stored value anywhere, you can configure another Function node that retrieves the value from the global context. Create an empty flow tab in the workspace, give it

a descriptive name, and place a function node and write this code inside:

```
msg.payload = global.get('temperature')
return msg;
```

7. Put an **`inject`** node upstream that sends a timestamp every 2 seconds and a debug node downstream for checking.

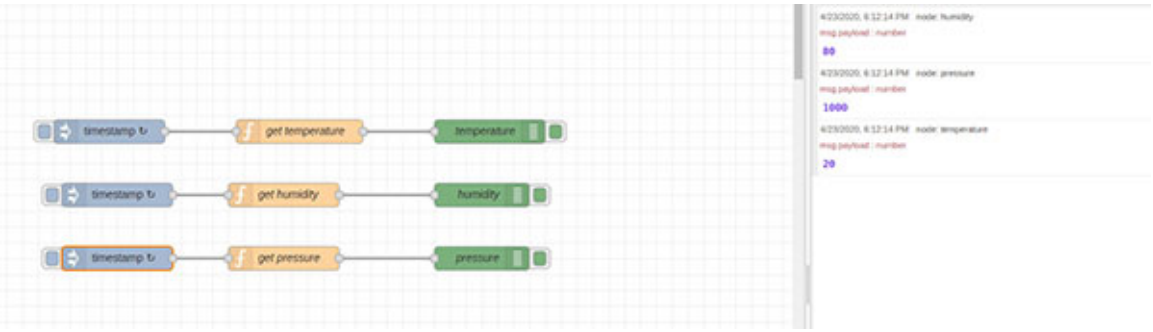


Figure 3.43: *Flow to retrieve the environment data currently stored in the global context*

8. Repeat this process for humidity and pressure variables, and you have finished.

You can find the context data stored in the user data path as defined in **`settings.js`** in the field userDir (data folder for our case). Access the content in the **`global.json`** file with the following command:

```
$ cat ~/book_hands-on-
iot/chapter3/app/data/context/global/global.json
```

The output should be like this:

```
{
  "temperature": 20,
  "humidity": 80,
  "pressure": 1000
}
```

You can see a JSON object that lists all the variables that you saved from the flow.

You are done, you have successfully completed the data acquisition functionality of your app, and have made available the environmental data to the outside world via a Pusher channel, fulfilling the emitting part of the streaming functionality.

Conclusion

In this chapter, you have completed the core software that will run in the IoT device, i.e., the Raspberry Pi. On top of it, you will plug in the Sense Hat that holds the sensors which will continuously be providing temperature, humidity, and pressure readings.

Hence, the software you have been running in the laptop is almost the same you will load into the Raspberry Pi, with the only difference that you will have a physical Sense Hat instead, and will have to use different NodeRED nodes, i.e. **https://flows.nodered.org/node/node-red-node-pi-sense-hat**, to acquire actual sensor data.

So far, you have got familiar with the virtual Sense Hat board, and learned how to create a NodeRED project. Then, you have integrated the external streaming tool called Pusher, and finally learned to store in NodeRED the current readings of the Sense Hat.

In the next chapter, we will progress to the remote server, where we will be receiving environmental data via Pusher and you will learn how to perform basic statistics with them. You will be introduced to the topic of how to store time series of the environmental variables, i.e., temperature.

Points to remember

- The Sense Hat is an *all-in-one* board for the Raspberry Pi to easily experiment with common sensors used in IoT and transport applications.
- The IoT Foundation Layer includes the data acquisition and broadcasting of sensor data using real-time streaming channels.
- A NodeRED project allows the development of an application in an isolated environment, providing portability to deploy it afterward in

production infrastructure.

- Streaming is a communication mechanism in which the client does not have to send requests to a remote server because the channel is permanently open and the client keeps listening for events as they are produced.

Multiple choice questions

1. **What is the Sense Hat?**
    a. a printed board circuit to attach to the Raspberry Pi
    b. an add-on board with sensors to attach to a PC
    c. a general-purpose add-on board with sensors and a LED matrix to use with the Raspberry Pi
    d. an all-in-one IoT device
2. **What is the main difference between the virtual Sense Hat and the actual one?**
    a. The virtual one produces fake sensor data.
    b. The virtual one can only be used with the PC, not with the Raspberry Pi.
    c. The cost of the actual Sense Hat is twice that of the virtual one.
    d. The LED matrix of the actual one is made of 10×10 LED.
3. **Out of the three numeric codes that designate the version of a NodeRED package, 3.2.5, which of them does designate a release that solves bugs of the package?**
    a. the code 3
    b. the code 2
    c. the code 5
    d. none of them, a package of the npm registry is rejected if it contains bugs
4. **What features provide a NodeRED project?**

a. a NodeRED package for your project that you can register in npm
b. a set of NodeRED nodes that provide a solution for an IoT problem
c. a git repository to track changes in the project
d. an encapsulated environment for your development, isolating the flows from other projects you are working on

5. **Why do you need to use secret keys in real time apps like in the case of Pusher?**

a. because of security reasons, to make sure that only people with which you share the app can access the data your device is publishing
b. because of backup reasons, to make sure that your customers may save their streamed data in Pusher
c. because of workload control reasons, to reduce the total amount of devices that can be listening to your channel
d. because of storage reasons, to save the data produced by sensors

Answers

1. **c**
2. **a**
3. **c**
4. **d**
5. **a**

Questions

1. List the sensors with which the Sense Hat is equipped and the type of data each one provides.
2. List the layers of the application and find where the data acquisition functionality is located.

3. Taking the point of view of NodeRED, explain what the software interface between the data acquisition and real-time streaming functionalities consists of.

Key terms

- **IoT foundation layer**: This is the basement layer of the software architecture where the physical hardware is located. It includes data acquisition from sensors as well as their broadcasting for processing in remote servers.
- **Data acquisition**: It is the functionality of the application that allows collecting measurements from sensors in a structured manner.
- **Streaming acquisition**: It is the functionality of the application that provides data broadcasting to make them available to potential consumers.
- **IMU**: It is the acronym for Inertial Measurement Unit, the assembly that includes an accelerometer, gyroscope, and magnetometer to provide an absolute orientation of a body in the 3D space.
- **Virtual sensor**: It provides a software-based twin of the physical sensor that emulates its behavior, both for the design and testing purposes of an IoT application.

CHAPTER 4

Real-time Data Processing with NodeRED

Introduction

This chapter builds the *data processing* functionality located in the *middleware layer*. It will be developed in your Linux laptop, while the final solution will be integrated as backend software in a Linux remote server.

Starting from the point in which we left the application in the previous chapter, we will build several NodeRED flows that will provide us valuable insights from the raw data supplied by the environmental sensors, as well as get familiar with a simple approach to save time series in a physical file.

To do so, we will cover several steps: design software architecture, calculate the statistics of the temperature data in real-time, and finally learn to store in a physical file the time series the of temperature data coming from the Sense Hat.

This chapter will provide you with the foundation to create a backend application that uses data provided by IoT devices and sensors.

Structure

In this chapter, we will discuss the following topics:

- Subscribing to Pusher channel for environmental conditions
- Software architecture for the application
- Basic statistics of environmental conditions
- Store aggregated data for later processing

Objectives

After studying this unit, you should understand how a data stream can be processed to obtain results in real time. You should also be able to acquire data using Pusher, as well as calculate basic statistical parameters of the stream, and store these results in a file database for post-processing and analytics.

Technical requirements

The code for this chapter is hosted at the GitHub repository that you can find at **https://github.com/Hands-on-IoT-Programming/chapter4**. In order to get your local copy, go to your home path and clone the repository:

```
$ cd ~/book_hands-on-iot
$ git clone https://github.com/Hands-on-IoT-
Programming/chapter4.git
```

Hence, change to the path of NodeRED app and install the dependencies defined in `package.json`:

```
$ cd ~/book_hands-on-iot/chapter4/app
$ npm install
```

This chapter's code provides several start options for NodeRED following the same approach explained in the clock widget exercise covered in *Chapter 2, Getting started with NodeRED*.Hence, the script section of `package.json` of the present code is as follows:

```
"scripts": {
  "start": "node node_modules/node-red/red.js --
  settings=./settings.js",
  "flow0": "node node_modules/node-red/red.js --
  settings=./settings.js flows/flow0.json",
  "flow1": "node node_modules/node-red/red.js --
  settings=./settings.js flows/flow1.json",
  "flow2": "node node_modules/node-red/red.js --
  settings=./settings.js flows/flow2.json"
}
```

Next, we will list the commands for every script, explaining what each one does:

- `npm start`: It launches NodeRED with a blank canvas to start from scratch.
- `npm run flow0`: This version contains the flows at the state in which they were at the end of the previous chapter. The setup is covered in the next subsection, and it is the point from which the practice will start.
- `npm run flow1`: This version of flows corresponds to section *Basic statistics of environmental conditions* below.
- `npm run flow2`: This version matches the result of section *Store aggregated data for later processing* below.

The next step has to be performed for every of the flow versions listed above, since the credentials are not stored in the descriptive JSON file for security reasons.

New NodeRED packages

In this chapter, we will introduce three new packages; each one with a specific goal as explained below:

1. Measure the interval between messages using the package `node-red-contrib-interval-length`, described at **https://flows.nodered.org/node/node-red-contrib-interval-length**
2. Build a buffer with the incoming data over which we will perform statistical calculation, `node-red-contrib-buffer-array`, the description for which is found at **https://flows.nodered.org/node/node-red-contrib-buffer-array**
3. Carry out basic statistics over the array built in the buffer. The package is called `node-red-contrib-statistics` and its URL is **https://flows.nodered.org/node/node-red-contrib-statistics**

Note: These dependencies ship with the code inside the file package.json, so that you do not have to install them by hand from the NodeRED editor. Hence, when you run npm install for the first time from the ./app folder, they will be installed. Alternatively, you could install them by hand from the NodeRED editor. Recap that the how-to was explained in the previous chapter, at the beginning of the section: Setting up a virtual Sense Hat in NodeRED.

Once we have in place the required dependencies, we can launch any of the versions of flows mentioned above:

```
$ npm run <FLOW_VERSION>
```

We will use each one in dedicated sections of the chapter. Now, let's proceed to complete the flow configuration.

Setup Pusher nodes to point to your account and app

Run the initial version of the flows from which to start building the statistical calculation for the environmental data:

```
$ npm run flow0
```

Then, visit `http://localhost:1880` and log in as usual, i.e., with username 'admin' and password 'raspberry'.

The initial flows that will be loaded are those from the NodeRED project of the previous chapter regarding the Sense Hat simulation and data streaming to Pusher.

Since authentication data in nodes is not stored in the flow file (`flow0.json`) for obvious reasons, you have to re-enter the credentials for both `pusher in` and `pusher out` nodes. For this point, we recap here the correspondence between Pusher credentials and fields in the node form. You can see them both side to side in the following figure:

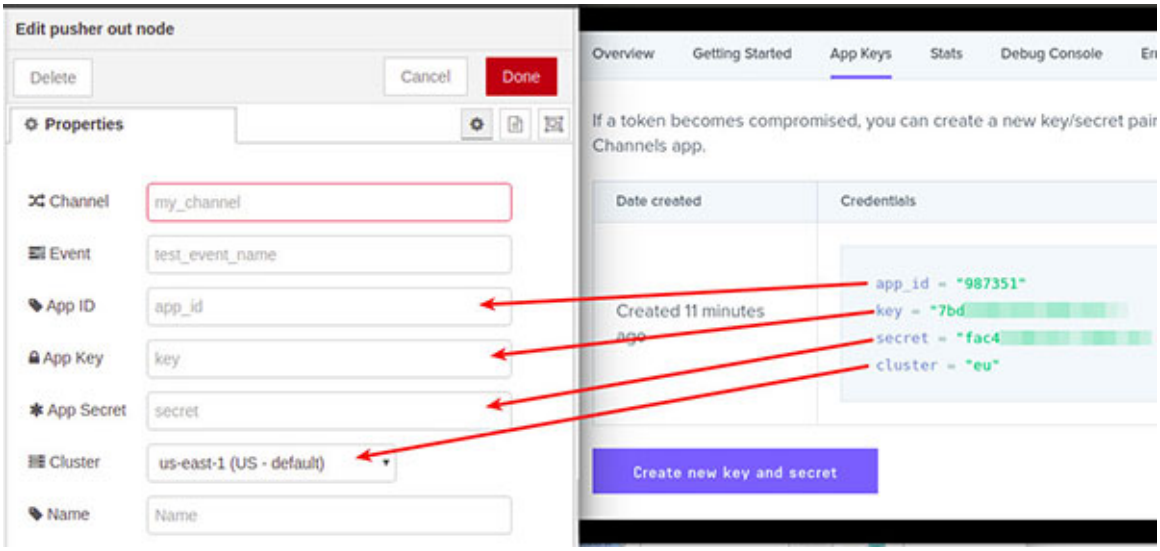


Figure 4.1: *Copy and paste Pusher credentials to NodeRED*

Hence, the filled form of the **`Pusher out`** node should appear as shown in the following figure:

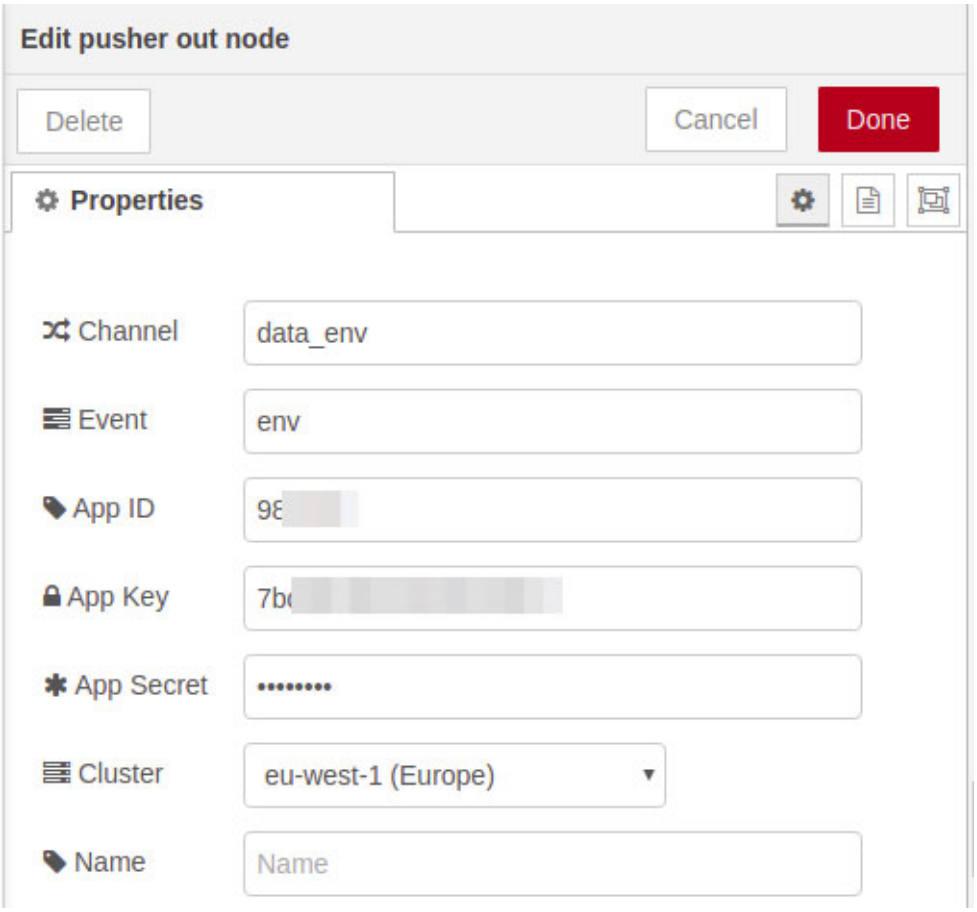


Figure 4.2: *Pusher out node properly configured*

For the `Pusher in` node, we also recap the steps:

1. Fill in the `App Key` and `Cluster` fields from the Pusher app credentials box.
2. NodeRED internal fields `Channel` and `Event` should coincide with the names you gave then in the pusher out node. `Channel` field value is `data_env` field and `Event` field value is `env`, as they were set in the pusher out node above.

In the following figure, you can see the proper configuration for `Pusher in` node:

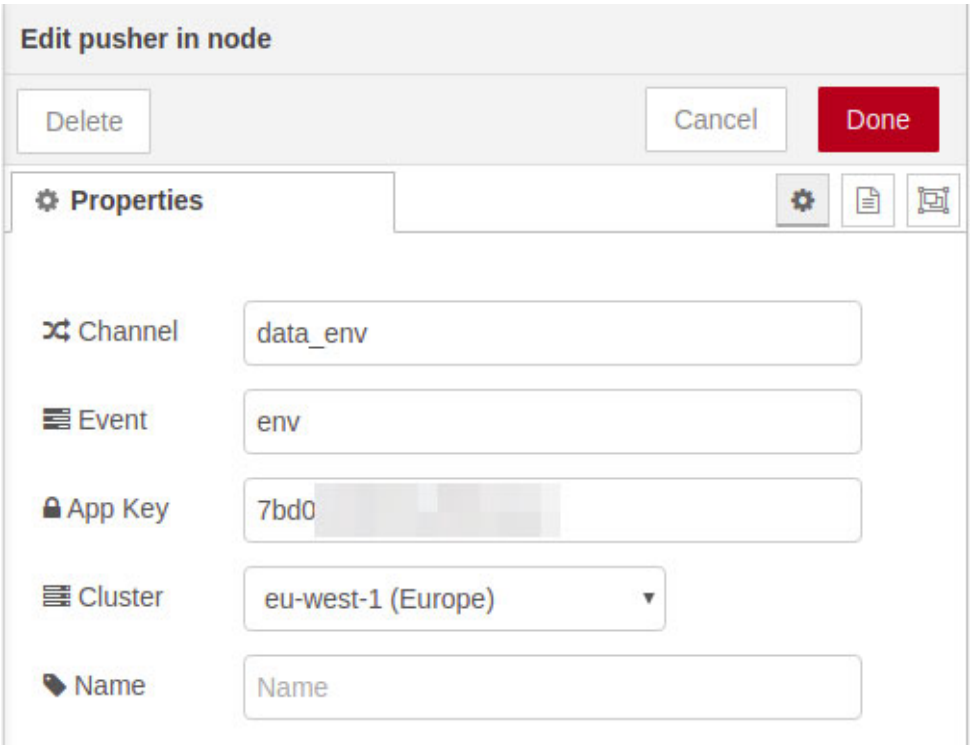


Figure 4.3: *Pusher in node properly configured*

Note: You have to follow the same process for every version of the flows supplied with the code, i.e., flow1.json, flow2.json and flow3.json. Once you set up the credentials, they will be stored encrypted in the NodeRED user data folder under a file called <NAME_of_FLOW>_cred.json, and you will not have to re-enter them again. Remember that this folder is created the first time you run NodeRED from the cloned repository (also recap that the name

and location is specified in the option userDir: '<PATH>' of the setting.js file).

Having completed these steps, you are ready to follow the practical exercises that we will cover in the next sections.

Subscribing to Pusher channel of environmental conditions

This section starts from the end of section *Publishing environmental data from the Sense Hat* of *Chapter 2, Data acquisition and real time streaming*, where we created the simple flow you can see in the following figure:

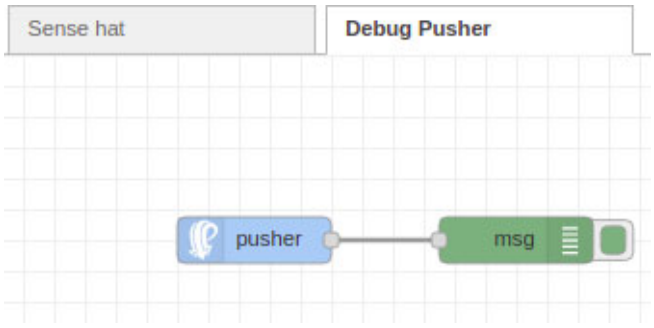


Figure 4.4: *Sample flow to check that the Pusher channel transmits the environmental data*

This flow listens to the Pusher channel, where we are sending the Sense Hat updates, and prints the incoming messages in the `Debug messages` tab of the Side bar.

You can bring it to life by launching the initial version of the flows:

```
$ npm run flow0
```

Then, visit `http://localhost:1880` and log in as usual. The Pusher in node was configured to connect to the environment app defined in Pusher dashboard, i.e., **https://dashboard.pusher.com**, and listen to events with topic name 'env'.

In the next section, we will wire several new nodes in order to make basic statistics of the incoming stream, i.e., mean and standard deviation.

Software architecture for the application

Before building the new flow, let's organize NodeRED tabs in a logical manner so that it matches the architecture of the application. To do so, you only have to rename the tab `Debug Pusher` to `Basic Statistics` and move it to the end, after the tab global context. The result should be similar to that in the following figure:

Figure 4.5: *Organization of flows in the NodeRED editor*

We do so to introduce a good practice that consists of organizing the flows in the order in which the data processing should occur. In this specific case, the order will be:

- `Sense Hat` tab: Acquire environment data and stream them to Pusher.
- `Global context` tab: Save current conditions to NodeRED context.
- `Basic statistics` tab: Subscribe to Pusher channel (event/topic env) and perform the statistical calculation to get insights from data in real time.

At this point, you should remember that until we enter into part III, *Hands on IoT Programming*, we will develop all the software in the laptop. Hence, it is important to mention how we should split the NodeRED application once we include the actual Raspberry Pi:

- Since data is transmitted via Pusher, we can separate flows occurring in the IT device (Raspberry Pi) from the rest, which can be hosted on a remote server. Hence, we can split tabs in two groups as shown in the following figure:

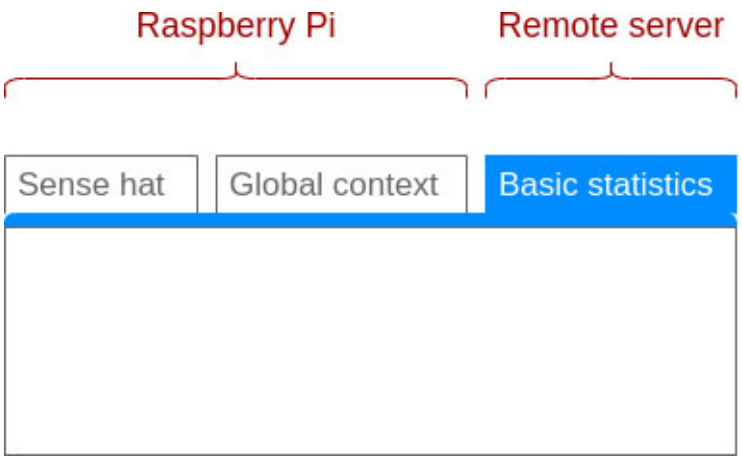


Figure 4.6: *Separation of flows between Raspberry Pi and remote server*

- The first two will be part of a NodeRED instance running in the Raspberry Pi, to which the Sense Hat will be physically attached.
- The last tab, i.e., Basic statistics, will be hosted in another NodeRED instance running on a remote server. This way, we can remove the calculation processes from the Raspberry Pi and offload its CPU.

For the scope of this chapter, we will keep all of them in the same NodeRED instance in order to avoid the complexity of having to manage several instances. In the last part of the book, you will perform the physical separation of the flows, keeping them connected via Pusher.

Basic statistics of environmental conditions

This section illustrates how you can process a stream of data and get real-time insights. This is something you usually perform on static data (i.e., an Excel spreadsheet) when you receive a table with the historical records of environmental conditions, import them into the spreadsheet, and calculate mean and standard deviation for a given time span.

This section takes you one step forward by dealing with the situation in which the data is not *static* but dynamic, i.e., there is a continuous stream coming from a sensor. This will let us confront the two ways of approaching decision-making:

- Review data and make *decisions based on a dashboard* built using recorded data. With this approach, there will always be a delay between data acquisition and decision-making.
- Take *decisions in real time*, since the dashboard is depicted and instantly updated as new data comes in. This approach also open the possibility to automate some decision processes that could be modelled with simple rule chains (i.e., conditions of the type if … then … else …)

In the dynamic and changing world we live in today, the first strategy is losing ground in favor of the second, something that is especially true for large enterprises, but not only. And that is because technology is providing such capability at a very tight cost. Bringing this reflection to our scope as developers or makers, the cost means just the laptop and the time we invest to build the app. Sounds so easy? Let's see next.

Note: To access the finished flows of this section, run the command npm run flow1, then visit http://localhost:1880 and log in as usual, i.e., with username 'admin' and password 'raspberry'.

Building the data processing code

The flows for this section are arranged in three components, each linked to one of the three new packages:

1. Build a buffer with the incoming data to be processed downstream, functionality that is performed using the package `node-red-contrib-buffer-array`.
2. Perform basic statistics over the array supplied by the buffer, i.e., calculation of mean and standard deviation. For such a purpose, we will use `node-red-contrib-statistics`.
3. Measure the interval between messages, for which we will need the package `node-red-contrib-interval-length`.

The combination of the three will provide the final code for the `Basic statistics` tab. Now, let's learn how to make each of them.

First flow: Build a buffer with the incoming data

Visit the `Nodes Palette` and drag and drop a node of type `buffer-array`. This node is placed under the function category. The only thing you have to configure for this node is the buffer size. Double click on it and set the length to 10 items. Then, place the node between Pusher in and the Debug nodes as shown in the following figure:

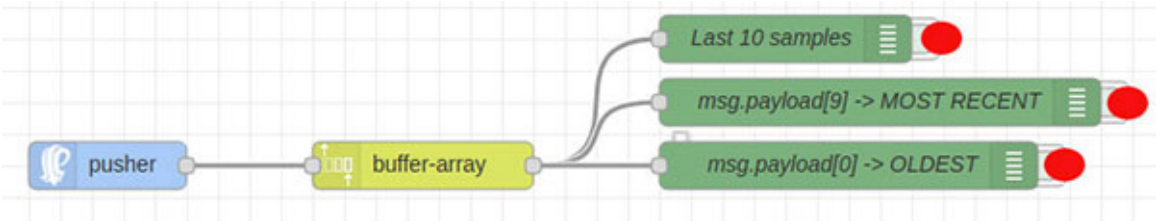


Figure 4.7: *Sample flow to explore the Buffer-array node*

Why are we using three debug nodes? Well, it's a didactic way to explain what the buffer is:

- As you have set its size to 10, the node will create an array of length ten and all items equal to 0.
- As new messages are received, the node will push their `msg.payload` content into the array starting from the last position, i.e., that with the index 9. The non-filled positions will remain equal to 0.
- When the second message comes in, item of index 9 will move to index 8 and the payload of the new message will be placed at index 9 of the array.
- For every new incoming message, all the current items will move one position up, leaving free the position of index 9 for the payload of that new message.
- When 10 messages have been received, the first one will be in the position of index 0. Hence, the next message will produce that it is dropped off, and index 0 will be occupied by the second message.
- Following this algorithm, you will always keep in memory the 10 most recent messages.

Note: In queuing theory, this structure is known as FIFO, that is the acronyms First In, First Out. This kind of queue is the most common in social situations, for example when you are going to pay in the market for the products you have in your basket. The first that comes to the queue is the first that is served.

Now you may guess the Sense of the three debug nodes:

- The top one, called Last 10 samples, will show the full array.
- The middle node, called *MOST RECENT*, will show the item in the position of index 9 of the array.
- The bottom one, called *OLDEST*, will show the item in the index 0 of the array. Hence, the debug expression has to be `msg.payload[0].temperature`

Note: The expression you have to specify in the Output field of the node is msg. payload[9].temperature. We select the attribute temperature to focus our analysis on one of the three values that come in each message (the other two are humidity and pressure).

Let's see the flow in action:

- The output of the top debug node is as shown below. You can see how the array is being filled from bottom to top as a new message comes, as shown in the following figure:

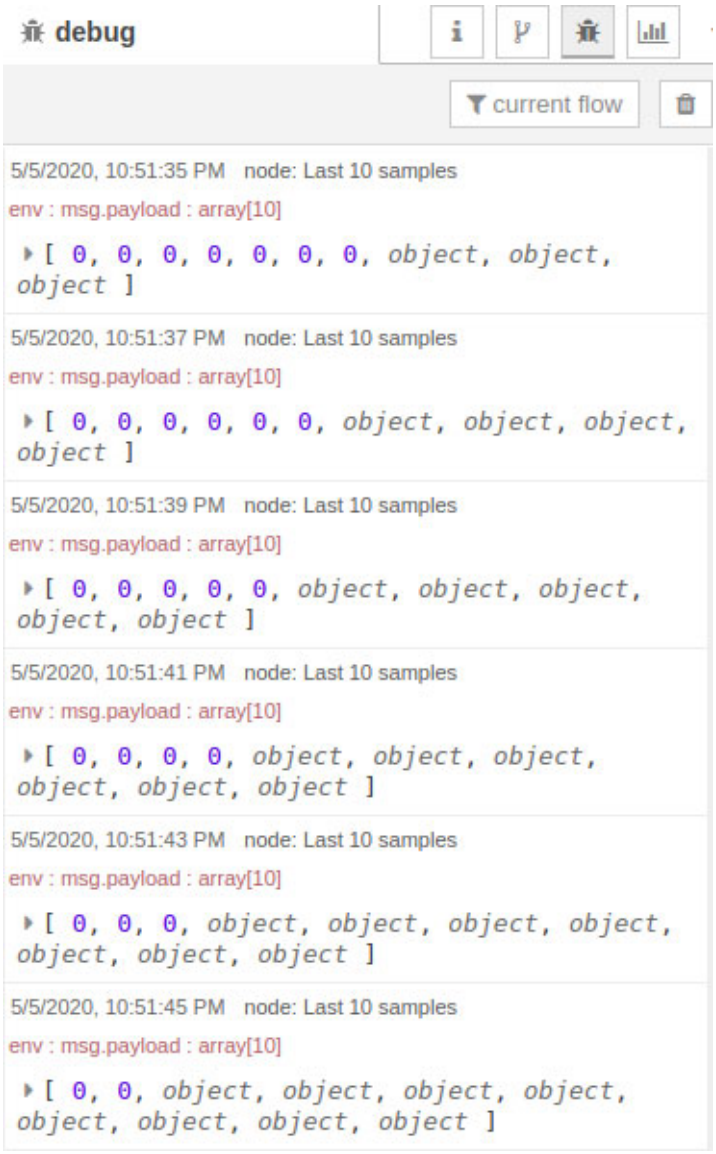


Figure 4.8: *Output from the buffer-array node*

If you expand the first message in the image above, you will see the composition of the object items as shown in the following figure:

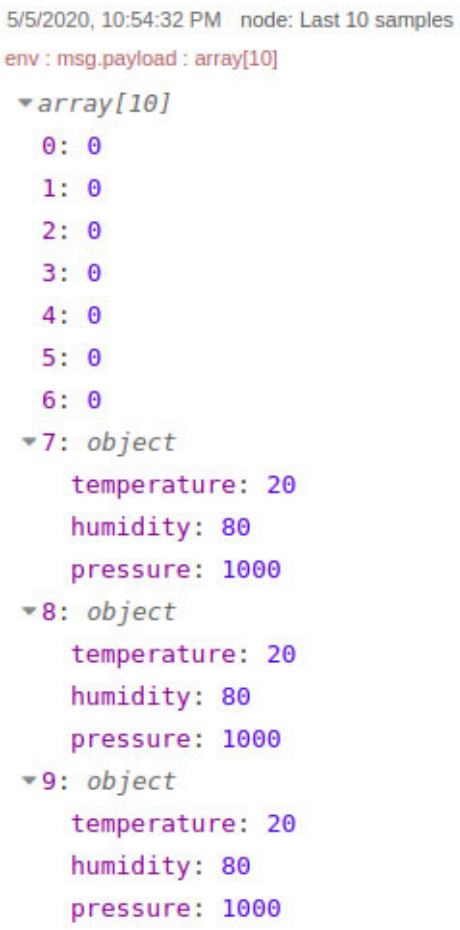


Figure 4.9: *Detail of the first non-zero items from the output of the buffer-array node*

As expected, every message arrives with the three values that characterize the environmental conditions at every instant.

- The output of the middle debug node will print for every occurrence the attribute temperature of the message is in the position of index 9.
- The output of the bottom debug node will print for every occurrence the attribute temperature of the message is in the position of index 0.

At this point, you should realize that the array contains the data we will use to perform basic statistics of the environmental conditions. Hence, we will always be reporting the mean and standard deviation of the last 10 data points.

Second flow: Perform basic statistics

Given that you already know how to store in an array the last 10 data points, we are ready to include the node provided by the package `node-red-contrib-statistics`. Drag and drop two nodes of type Statistics from the Nodes palette. This node type is also placed under the function category. The flow should look like the following figure:

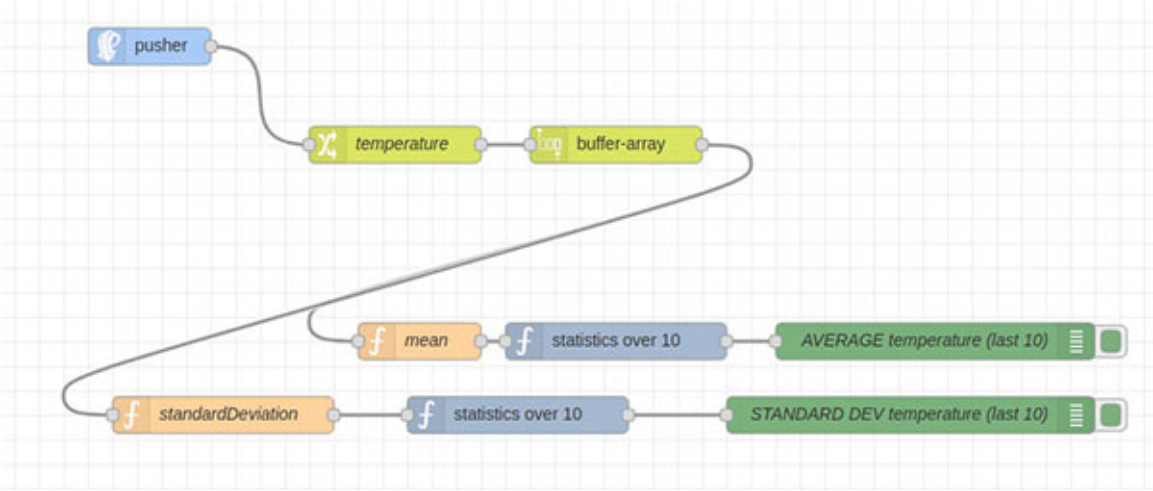


Figure 4.10: *Sample flow to explore the Statistics node*

The `temperature` node is of Change type, and allows reconfiguring the payload to keep only the temperature data as shown in the following figure:

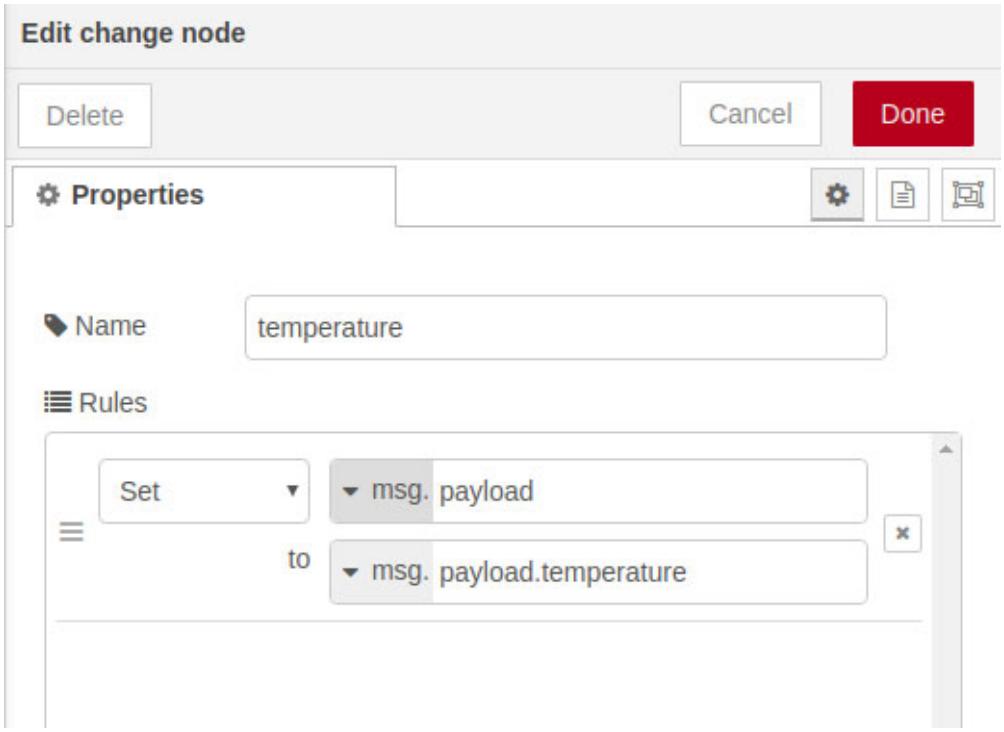


Figure 4.11: *Configuration of the Change node for extracting temperature values*

The two **`Statistics`** nodes only need to configure the **`Data Set Size`** field. Double click on each of them and set such value to 10, as your buffer node will create an array of that size.

Finally, the function node preceding each statistics node specifies the calculation to perform:

- For the average value, the body of the **`mean`** function node has to be:

```
msg.topic="mean";
return msg;
```

- For the standard deviation, the body of the **`standardDeviation`** function node has to be:

```
msg.topic="standardDeviation";
return msg;
```

The last flow of the current development corresponds to the package **`node-red-contrib-interval-length`**. This will give us the chance to create a health indicator of our application.

Third flow: Measure the interval between messages

Drag and drop one node of type `Interval length` from the Nodes palette. This node type is placed under the function category. In principle, you don't need to configure anything, just wire it between Pusher in and debug nodes as shown in the following figure:

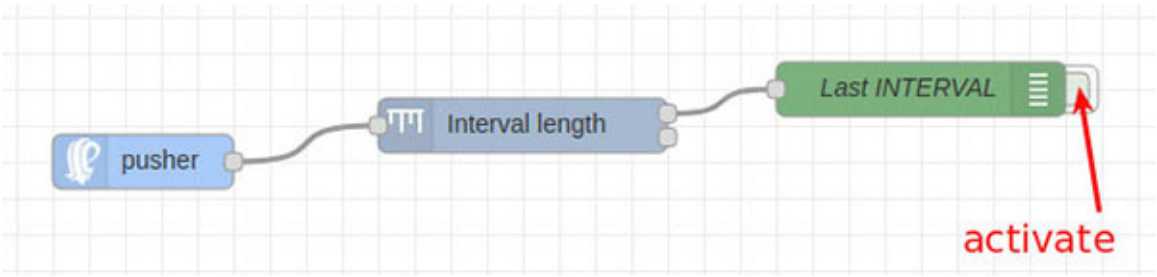


Figure 4.12: *Sample flow to explore the Interval length node taking the Pusher stream as input*

When you activate the debug node, the `Debug messages` tab of the side bar will show something like the following figure:

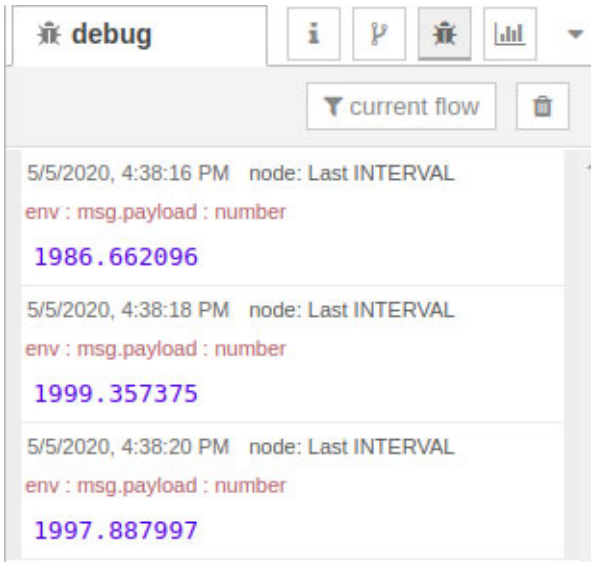


Figure 4.13: *Output from the Interval length node*

The result is expressed in milliseconds, and the node computes the interval between consecutive messages. Theoretically, it should always be 2000 ms, i.e., the frequency that we are sending the data to Pusher

from the `Sense hat` tab. Why doesn't it exactly match that value every interval?

To obtain the answer, let's apply the same interval calculation to the data stream just before being published to Pusher. Copy and paste the two nodes to the `Sense hat` tab and wire them just after the throttle node that delivers messages at the required rate of 1 every 2 seconds. You can visualize this part of the workflow in the following figure:

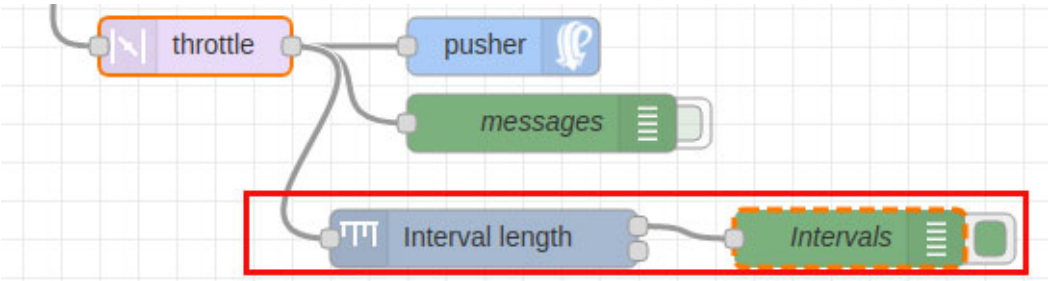


Figure 4.14: *Sample flow to explore the Interval length node taking as input the Sense Hat*

Finally, deploy the flows and compare the `Debug messages` tab of `Sense hat` (right side in the image below) and [aux] `interval length` tab (left side). You should find something like the following figure:

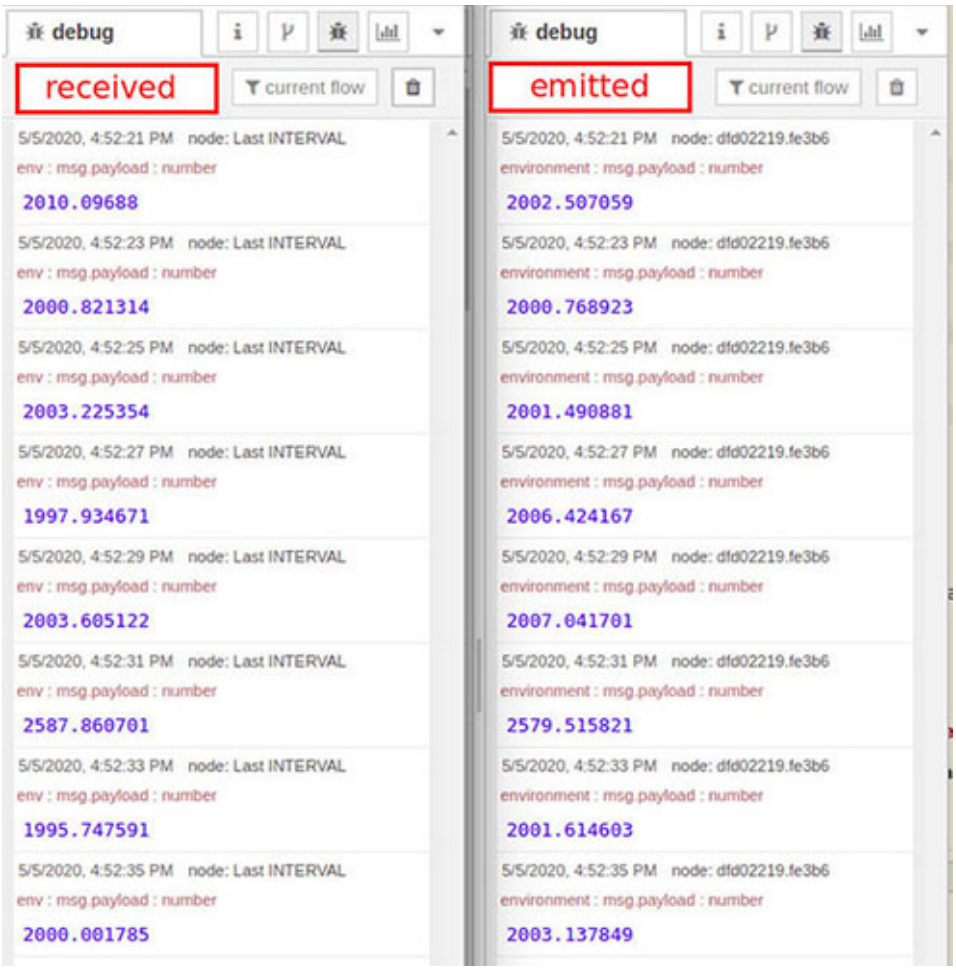


Figure 4.15: *Comparison of the outputs' intervals: Sense Hat vs. Pusher stream*

It can be appreciated that some minor errors are introduced in both stages:

1. In the emission stage, before delivery to Pusher: This error is due to the throttle node and it is of the order of 10 ms (deviation from 2000 ms).
2. In the reception stage, when receiving from Pusher: This error can be estimated by subtracting the emitted interval (right side) from the received interval (left side). The list of errors in milliseconds is 7, 0, 2, -3, 8, -6, -3. If you sum all of them, the result is 4, and if you include many more differences, their total sum tends to zero.

This cancelation occurs because the interval differences compensate for themselves. To understand why, think of a stream of 10 messages that are delivered in a total span of 20 seconds:

- The stream of messages transmitted by Pusher will also be distributed in a span of 20 seconds.
- Since the input stream covers 20 seconds, and the output stream frequency is also 20 seconds, the interval differences necessarily have to compensate for each other. If it were not the case and the errors keep accumulating, it would imply that the output stream will take more than 20 seconds to deliver the 10 messages, let's say 20.8 seconds. That would mean receiving each new message with an increased delay, and after many messages, we could have accumulated a delay of seconds, and even minutes. But it is not the case: streaming services like Pusher base their value on delivering real messages without introducing significant delays.

Has this minimal interval error impacted our application? Definitely not, because we measure environmental conditions, and their changes over time are orders of magnitude greater (minutes and hours), while the interval errors are in the order of milliseconds.

This conclusion suggests a final question: why are we interested in calculating the intervals between messages? The answer is that this is an indicator of whether the data chain is working properly. We call it chain because there are two possible points of failure whose effects are additive:

1. **Data generation**: for example, the physical sensor is not working properly; sometimes the messages are delivered every 2 seconds, and when failing, a message might be delivered 5 seconds after the previous one.
2. **Data transmission:** in the real-time layer provided by Pusher. If this service were not in a healthy state (for example, many different users overloading Pusher servers), we could see larger differences in the interval lengths, in the order of the emission rate (2 seconds).

So, it should be considered as a *health indicator* of our application. We will consider that the *service is healthy if the interval lengths are at least one order of magnitude lower than the emission rate, i.e., 0.2 seconds*.

To finish this subsection, let's recap putting together these three functionalities. Since they share the same input, the Pusher in node, we can include them in a single flow to achieve a more compact code. The result is shown in the following figure:

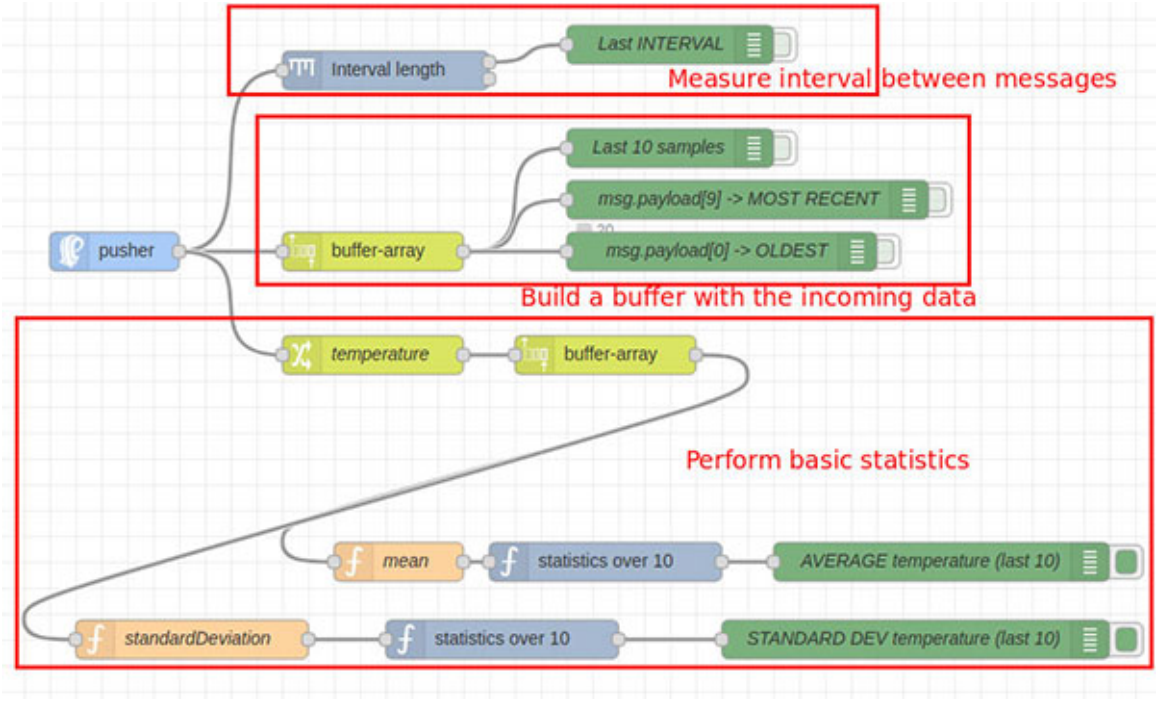


Figure 4.16: *Putting together the three functionalities*

This is what the `Basic statistics` tab shows. The next step is to perform a functional test of the flow to check whether it works properly.

Test the flow

The basic idea is to design a simple test that lets you guess that the statistical calculations are right. It follows a sample test scheme, but you are encouraged to design your own and pass it to the flow:

1. Find the virtual sensor at the URL **http://localhost:1880/sensehat-simulator**, and manually increase the temperature from 20ºC to 22ºC.
2. Take 10 consecutive measurements of temperature. Since we have one data point every 2 seconds, we will have to wait 20 seconds to get all of them.

3. After 20 seconds, the mean temperature will be equal to the setpoint (22ºC) and the standard deviation will be 0. This is because the last 10 measurements over which the calculation is performed are all equal to 22ºC.

Let's pass this test to the flow:

1. Set the virtual sensor to 22 Celsius degrees as shown in the following figure:

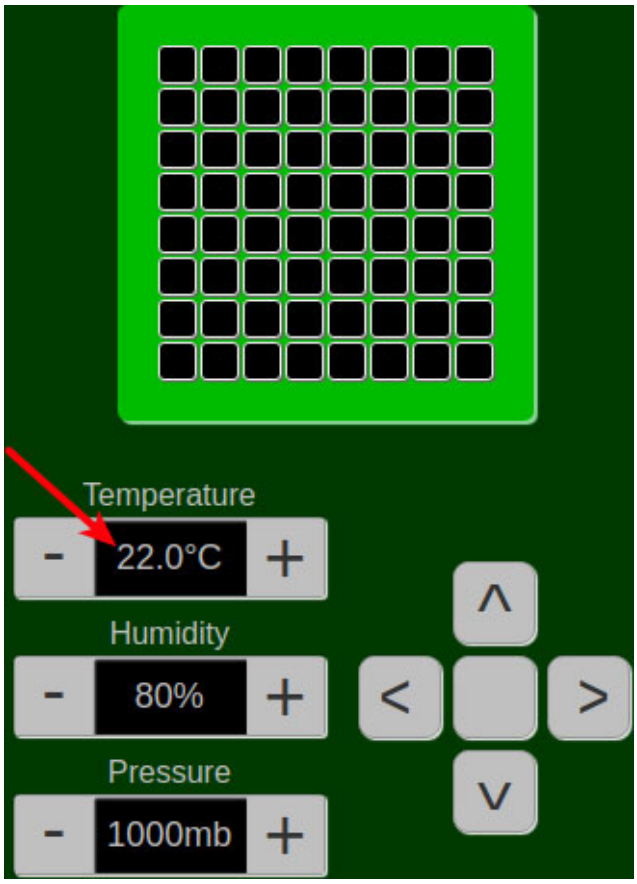


Figure 4.17: *Sense Hat Graphical User Interface*

2. Have a look at the `Debug messages` tab to find how mean and standard deviation change with every new point. Mean is increasing with every new reading, while standard deviation progressively decreases to zero.

Figure 4.18: *Output statistics of last ten measurements*

3. After 10 calculations (i.e., 10 measurements are finally available) find what mean and standard deviation are:

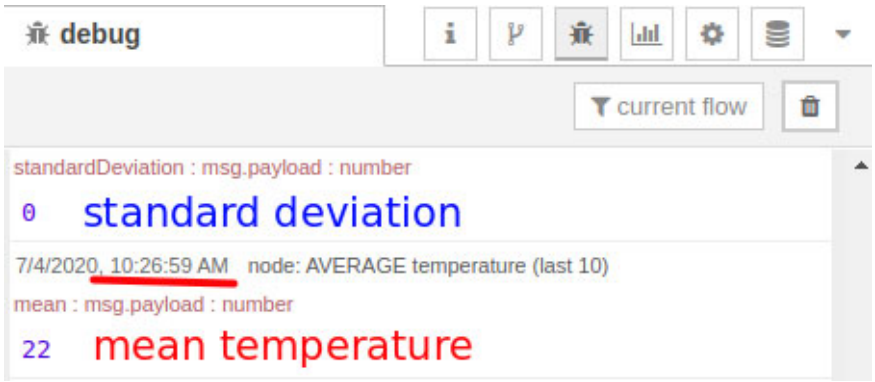


Figure 4.19: *Output statistics of the last ten non-zero measurements*

We find 22 Celsius degrees mean temperature, and 0 standard deviation because all 10 measurements have the same value (22).

We can make this convergence more evident by painting a graph of data and calculated statistics. Let's see how.

Graph the results

Let's take the set of nodes corresponding to the **`Perform basic statistics`** tab, and **`add a Chart`** node (part of the dashboard category) from the nodes palette. Also, we need to do some pre-processing to feed the data to the graph, since we would like to see the curves of instant temperature, mean and standard deviation together. The new nodes are enclosed within a red box in the next figure:

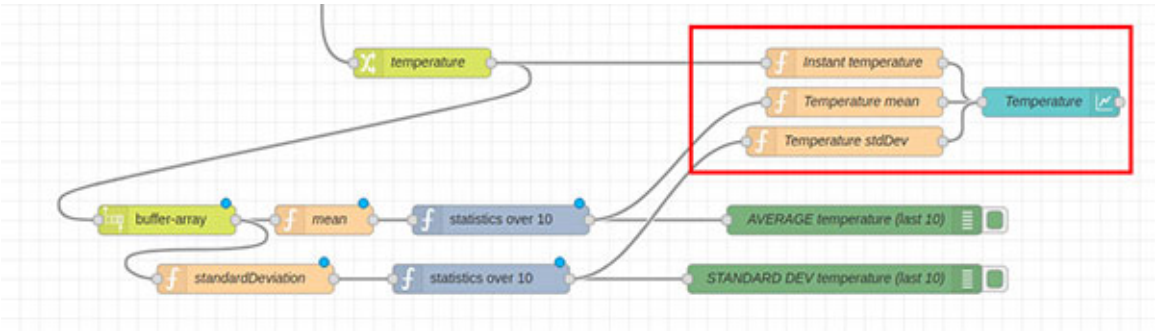


Figure 4.20: *Flow to graphically represent the time series of temperature statistics*

So that we can plot together the three curves, we have to take each payload—from nodes temperature and the two of statistics over 10—and give a different name to every **`msg`** object. Hence, the content of the three function nodes performs these operations:

- For the instant temperature, we define the **`msg1`** object:

```
msg1 = {payload: msg.payload, topic: 'Instant read'};
return [msg1];
```

- For the mean temperature, the **`msg2`** object:

```
msg2 = {payload: msg.payload, topic: 'Average over last 10
reads'}
return [msg2];
```

- For the standard deviation temperature, the **`msg3`** object:

```
msg3 = {payload: msg.payload, topic: 'Standard deviation over
last 10 reads'}
return [msg3];
```

What you specify in the topic field of each message, it is what will be shown in the graph as the legend of each curve.

Finally deploy the flow, open a browser window pointing to **`http://localhost:1880/ui`**, and pass the test as described above. You should see a graph as the one shown in the following figure:

Figure 4.21: *Plot of the time series of temperature statistics*

Over 20 seconds the standard deviation is not 0, while the mean temperature gradually goes from 20 Celsius degrees to the new setpoint at 22 Celsius degrees.

To end this chapter, let's see a quick way to store data from the sensor under the form of a plain *JSON* file. We leave a safer and a proper way to store the measurements using a database like *InfluxDB* for the next chapter.

Storing aggregated data for later processing

The simplest way to persist any kind of data is to store them in a physical file, and this is the approach we are going to follow to save the temperature time series. NodeRED provides a package for this purpose that is called `json-db-node-red`. In the next subsection, we will detail how to install it.

Note: To access the finished flows of this section, run the command npm run flow2, then visit http://localhost:1880 and login as usual, i.e., username 'admin' and password 'raspberry'.

Regarding the app structure, we will add a tab in the NodeRED editor as shown in this figure, where we will place the flow that will implement the file storage functionality:

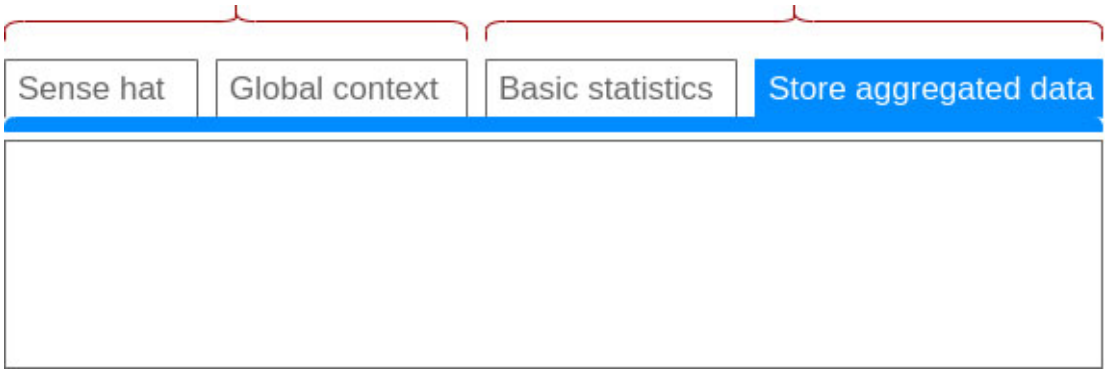


Figure 4.22: *Adding a tab for the file storage flow*

We can name this tab as *Store aggregated data*, as suggested above.

A package for storing data

First of all, we need to install a package that will allow us to persist data in a local file using the standard JSON format. This new module is called `json-db-node-red` and its NodeRED page is **https://flows.nodered.org/node/json-db-node-red**

To install it, follow the steps depicted in the image below:

1. Deploy the right menu (circle), click on `Manage palette`.
2. Select the `Install` tab and type in `json-db`-node-red (arrow) to find the package.

3. Then, press the `Install` button.
4. A pop-up window will appear with some information. Confirm the installation by pressing the right button.

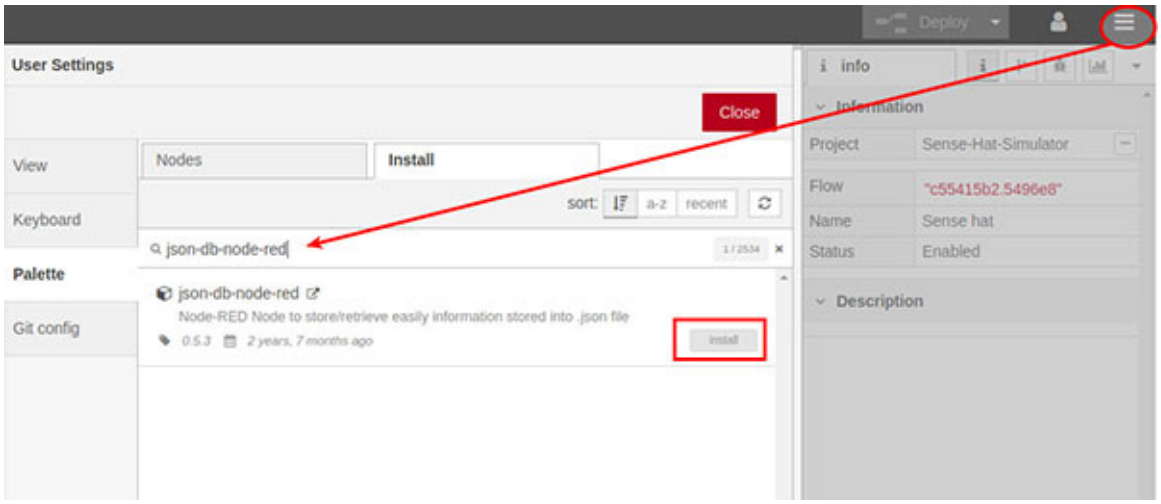


Figure 4.23: *Installing package json-db-node-red*

Once the installation is finished, look at the left pane to look for the new nodes. You can find them under the storage category: `DataIn` is to save data to a JSON file, while `DataOut` allows you to retrieve the information in a JSON file.

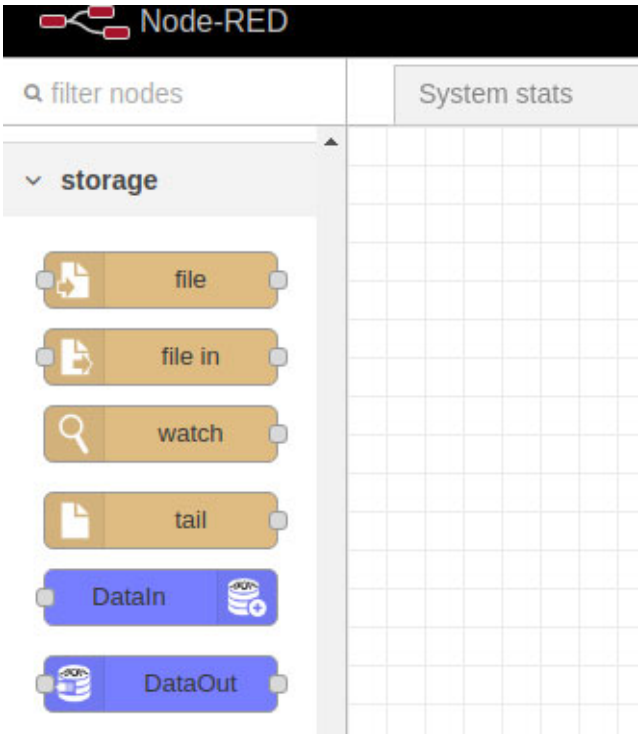


Figure 4.24: *Nodes of the json-db-node-red package*

We will make use of this new package to save the environmental data provided by the virtual sensing device, i.e., Sense Hat.

Storing the environmental data

Once the new package is in place, we have to forward the streamed data from the `Basic statistics` tab to the `Store aggregated data` tab. NodeRED provides an easy way to transfer messages from one place to another, whether the destination is in the same tab or not. If you type in the link in the search box of the Nodes palette, you will find a source node, Link out, as well as a destination node, Link in:

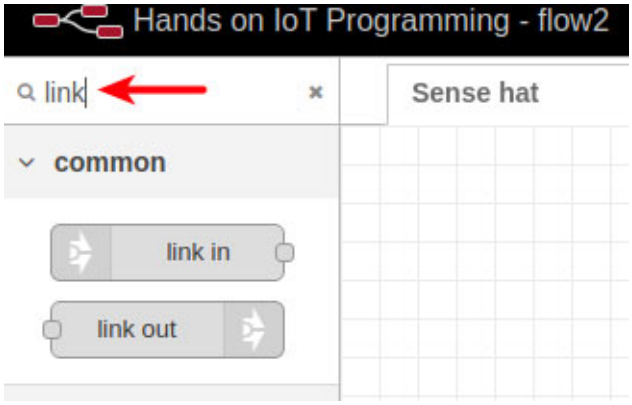


Figure 4.25: *Nodes to transmit messages between flows without explicit wires*

You should use the `link out` node for exposing the messages flowing through the incoming wire, and `link in` to receive those messages anywhere in the application.

So, put the `link out` node just after the `temperature` node in the `Basic statistics` tab, as shown in the following figure:

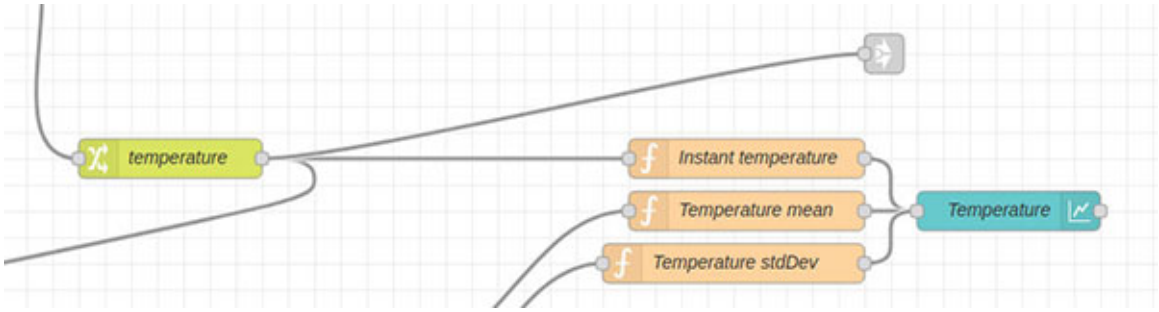


Figure 4.26: *Link out node to broadcast temperature*

Double click on it and give the name `Store temperature`, because in the next step you will have to refer to it so that the connection takes effect.

Now, switch to the `Store aggregated data` tab and place a `link in` node as shown in the following figure:

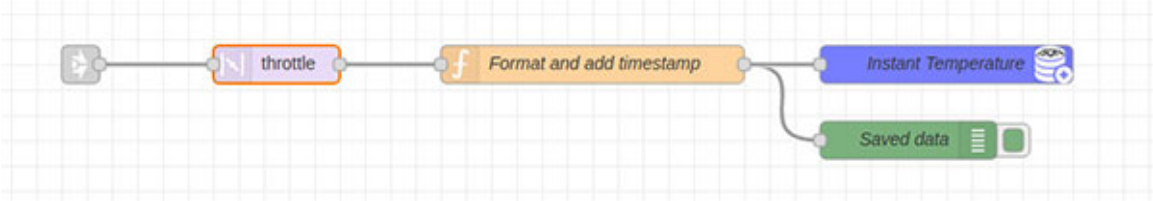


Figure 4.27: *Link in node to receive temperature*

Double click on the node and make sure to put a tick near the Store temperature name, that refers to the `link out` node you defined above as shown in the following figure:

Figure 4.28: *Editing the Link in node so that it connects to broadcaster Link out*

This way, both flows will be connected. Then, to build the new flow you need to add three nodes as follows:

1. **A Throttle node to save data points** at a given frequency: By double clicking on it, you can specify the interval at which you want records to be saved. For the purpose of this example, we are setting it to 10 seconds so that you don't have to wait to see how the time series is growing. But in a real scenario, you really don't need such a small interval because environmental conditions change slowly, in the order of minutes. Hence, it's good if you specify 2 or 5 minutes.

Figure 4.29: *Throttle node to save to data points*

2. A function node named **`Format and add timestamp`** whose purpose is to add a timestamp to every recorded point: In the right part of the figure shown below, you can see how the **`msg`** object is modified so that **`msg.payload`** carries all the information we need for every temperature record.

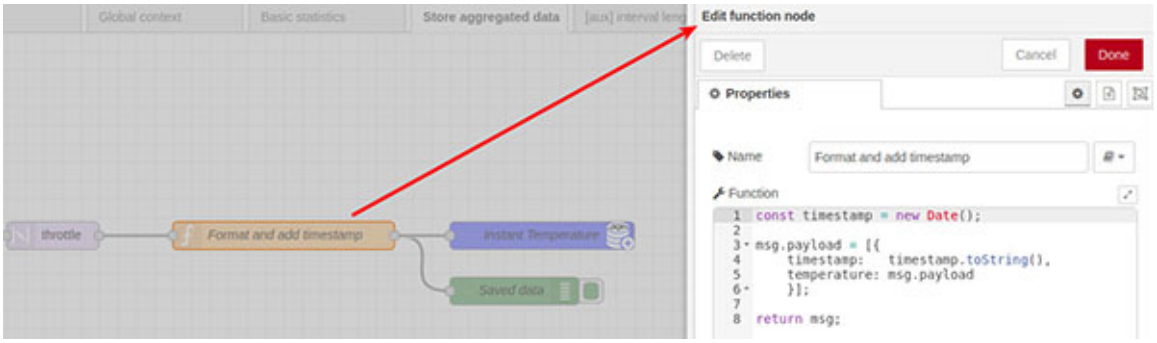


Figure 4.30: *Function node to add a timestamp to every recorded point*

3. A **`DataIn`** node named **`Instant Temperature`**, that belongs to the package we have just installed for storing data in a file as shown in the following figure:

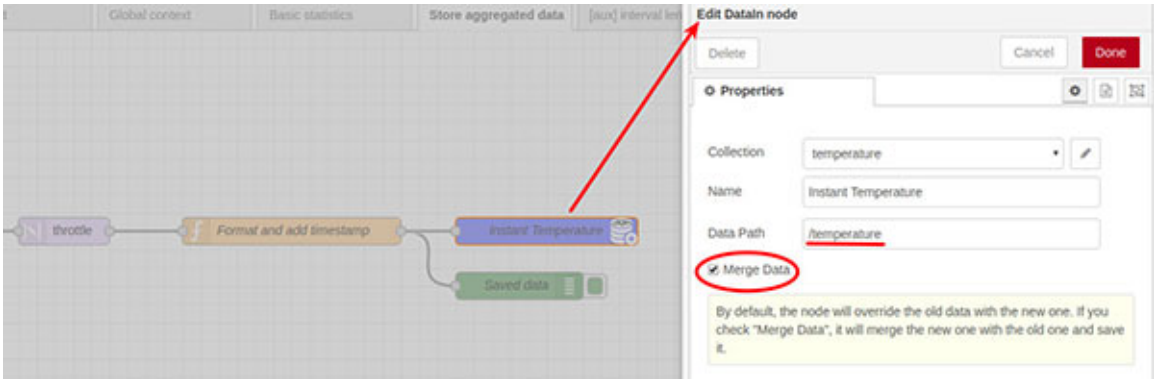


Figure 4.31: *DataIn node to store the measurements*

From the node form we have to create a collection accessing the editor by clicking on the icon. Its identification is provided in the field **`collection`**. In this case, it will be called **`instant-temperature`**, and hence the file name will be **`instant-temperature.json`**.

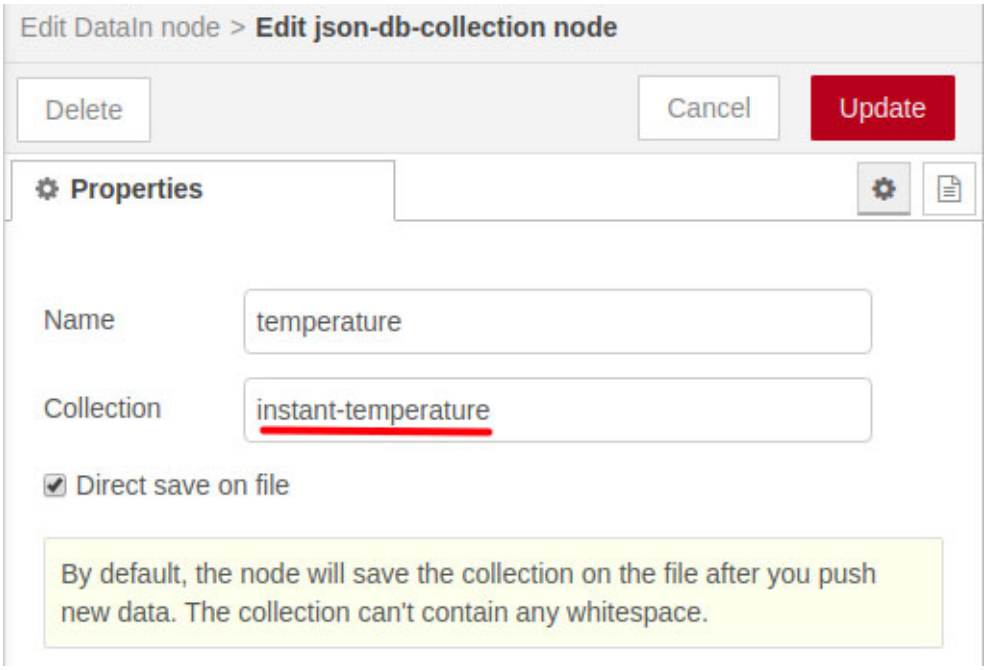


Figure 4.32: *Configuring the Collection of the DataIn node*

The file takes the name from the Collection name as specified in the form above. Such a file is placed in the folder where the user data for the application is stored, i.e., **`./data/JsonDB/instant-temperature.json`**.

Once the flow is deployed, if you take a look at that file, you will see a list of JSON objects under the **`temperature`** array. This array corresponds to the **`Data Path`** field you specified above in the Data in node form.

```
{
 "temperature": [
  {
   "timestamp": "Mon Jul 06 2020 19:52:48 GMT+0200 (Central
   European Summer Time)",
   "temperature": 20
  },
  {
```

```
    "timestamp": "Mon Jul 06 2020 19:52:59 GMT+0200 (Central
    European Summer Time)",
    "temperature": 20.8
   },
   {
    "timestamp": "Mon Jul 06 2020 19:53:11 GMT+0200 (Central
    European Summer Time)",
    "temperature": 21.6
   },
   {
    "timestamp": "Mon Jul 06 2020 19:53:23 GMT+0200 (Central
    European Summer Time)",
    "temperature": 21.6
   }
  ]
}
```

To make sure that you record a time series, you have to check the tick Merge Data in the node form. Otherwise, each new reading will replace the existing data and you will only have the current state, not the history.

At this point, you are recording dummy environmental data. Recall that in the last part of the book you will record real data of the environment where you place the Raspberry Pi.

Conclusion

In this chapter, we have provided a software architecture for our app that we made intuitive by separating the flows in NodeRED tabs: Sense hat, Global context, Basic statistics and Store aggregated data.

The processing these flows perform happens in the same order in which the tabs are organized: first, produce the dummy environmental data with a Sense Hat simulator; second, save current measurements in the NodeRED context; third, perform basic statistics over the time series; fourth and last, permanently store measurements in a physical file for later post-processing and analytics.

This way of organizing an application's workflow is known as a pipeline, and it is the most common structure when dealing with data from sensors.

In the next chapter, you will extend this pipeline to add persistent storage for your data in a remote database.

So far, you have become familiar with the structure of a simple backend application that uses data from IoT devices and sensors. You have also learned basic procedures to test NodeRED flows using graphical tools.

The next chapter focuses on how to store collections of data using databases safely. You will be introduced to the topic of the why and how to use databases. For the practical part, you will learn to work with MongoDB, the general-purpose database used in many web applications.

Points to remember

- Data processing in the application can be split in two parts: one in the IoT device (Raspberry Pi) and the other in a remote server.
- Communication between the IoT device and the remote server is possible thanks to data streaming using Pusher.
- Data processing usually takes a modular structure known as a pipeline, where the output of each step is the input for the next one.

Multiple choice questions

1. **Where are the credentials introduced in nodes stored?**
    a. within the file of the NodeRED flow
    b. not stored because it is not safe to have credentials available in a file
    c. has to be written each time you start the NodeRED application
    d. within the user data folder, in an encrypted file related to the flow file

2. **What does an Interval length node do?**
    a. obtain the average time interval between consecutive messages
    b. obtain the time interval between consecutive messages
    c. obtain the minimum time interval between consecutive messages
    d. obtain the maximum time interval between consecutive messages
3. **What does a buffer array node do?**
    a. build an array of incoming messages for persistent storage purpose
    b. build an array of incoming messages for volatile storage purpose
    c. build an array of incoming messages for downstream processing
    d. build an array of incoming messages for array operations
4. **What does a Throttle node do?**
    a. It delivers messages at a given rate to avoid data overload in the flow.
    b. It takes a stream of messages and delivers them at exactly the same constant rate.
    c. It completes the stream of messages with dummy ones if the rate is larger than the frequency at which they arrive.
    d. It accelerates the flow processing so that calculations end as soon as possible.
5. **What is the safest way to store data in an application from the point of view of their integrity?**
    a. using variables stored in the RAM memory of the laptop
    b. in a physical file
    c. in a local database
    d. in a remote database plus a local physical file

Answers

1. **d**
2. **b**
3. **c**
4. **a**
5. **d**

Questions

1. What are the modules of the pipeline that implement the NodeRED flows of the basic statistics of environmental conditions?
2. Explain the various generations of computers.
3. Explain the advantages of taking decisions based on dynamic data instead of static data.

Key terms

- **Middleware layer**: This is the layer that sits on top of the basement layer. It provides remote data processing and storage that will be served to the application layer.
- **Real-time streaming**: Functionality of an application that makes available data from sources that supply them in a continuous way, as is the case with IoT sensors. This functionality provides an interface for the consumers to use the data.
- **Data processing**: It refers to the sequence of operations and calculations performed on the data feed coming from the IoT sensor.
- **Pipeline**: Modular structure for processing data in which the output of each step/ module is the input for the next one.

CHAPTER 5

Storing and Graphing Data Streams with InfluxDB and Grafana

Introduction

In the previous chapter, we retrieved Sense Hat simulator data via the streaming application Pusher. This chapter naturally follows the pipeline and will cover the data storage functionality (in the *middleware layer*) and a dashboard visualization that will be accessible in the web browser. With Grafana, we will graphically represent the data stream, part of the *application layer*. It will be developed in your Linux laptop, while in the final solution it will be integrated as an application component in a remote server.

You will get familiar with the usage of databases while getting into the details of working with data coming from sensors.

Again, we will start at the point where we left the application in the previous chapter, adding new NodeRED nodes to store environmental conditions over time.

You will learn that the data persistence function has the goal of providing safe storage and offering quick access to stored data. And this is something that can be particularly tricky when dealing with time series, where you may have millions of points saved in the database. That is the reason for which, in this book, we opt for a database like *InfluxDB* that is properly designed to deal with this kind of data.

After covering all the sections in this chapter, you will acquire the insight of how the storage of data is relevant in order to build the visual dashboards for the users of your application.

Structure

In this chapter, we will cover the following topics:

- Software architecture for the application
- Setting up an InfluxDB database
- Storing processed data of environmental conditions
- Creating a visualization dashboard with Grafana

Objectives

After covering this unit, you should be able to create pipelines for raw data processing of sensor measurements, store data in an InfluxDB database, and integrate it with NodeRED. Finally, you will learn to quickly create a dashboard with an interactive visualization of time series of temperature.

Technical requirements

The code for this chapter is in the GitHub repository that you can find at **https://github.com/Hands-on-IoT-Programming/chapter5**

In order to get your local copy, go to your home path and clone the repository as follows:

```
$ cd ~/book_hands-on-iot
$ git clone https://github.com/Hands-on-IoT-
Programming/chapter5.git
```

Hence, change to the path of NodeRED app and install the dependencies defined in `package.json`

```
$ cd ~/book_hands-on-iot/chapter5/app
$ npm install
```

You should have an overview of the `package.json` file because it covers not only the state of the application for this chapter, but also allows you to directly access the versions developed in the previous chapter:

- `flow0-CH3`: This script launches the bundle of NodeRED flows at the state in which the application was at the end of *Chapter 4, Real-time data processing with NodeRED*.

- `flow1-CH4`: This version of the flows corresponds to the completion of the section *Basic statistics of environmental conditions* of the previous *Chapter 4, Real-time data processing with NodeRED*.
- `flow2-CH4`: This version matches the result of section *Store aggregated data* of the previous chapter, and represents its final version.
- `flow3`: This script hosts the state of the application as of the end of the current chapter.

You can run any of these scripts from the application folder with the already known command:

```
$ npm run <FLOW_VERSION_NUMBER>
```

Note: To access the finished flows of this section, run the command npm run flow3, then visit http://localhost:1880 and log in as usual, i.e., with username 'admin' and password 'raspberry'.

For the practical part of this chapter, you will need to install the nodes to store sensor data in the *InfluxDB* database.

New NodeRED packages

As with any relevant tool that is used for IoT applications, InfluxDB also provides a connector for NodeRED. The package name is `node-red-contrib-influxdb`, and it provides nodes for both saving and retrieving time dependent data from InfluxDB.

The package is described at **https://flows.nodered.org/node/node-red-contrib-influxdb**

Note: This dependency ships with the code of this chapter inside the file `package.json`. Hence, when you run `npm, install` from the `./app` **folder that you will be including these new packages.**Alternatively, you could install it by hand from the NodeRED editor. Recap that the how-to was explained in *Chapter 4, Real time Data Processing with NodeRED*, at the beginning of the section *Setting up a virtual Sense Hat in NodeRED*.

Once we have in place the required dependencies, we should start from the version of flows at the end of the previous chapter:

```
$ npm run flow2-CH4
```

The final step of the technical requirements of this chapter deals with connecting NodeRED to the Pusher account.

Setup Pusher nodes to receive the data stream

For this purpose, you have to follow the same procedure explained in the subsection with the same name, under section *Technical requirements* of the previous chapter.

Software architecture for the IoT application

In this chapter, we will add our third software component: first was NodeRED, then the streaming service Pusher, and now we will put on top the database, InfluxDB.

In the following figure, you can see the whole application structure as explained in *Chapter 1, Introduction to IoT applications and their Software Architecture*. The following figure lets you see in context the components that we will describe in this chapter:

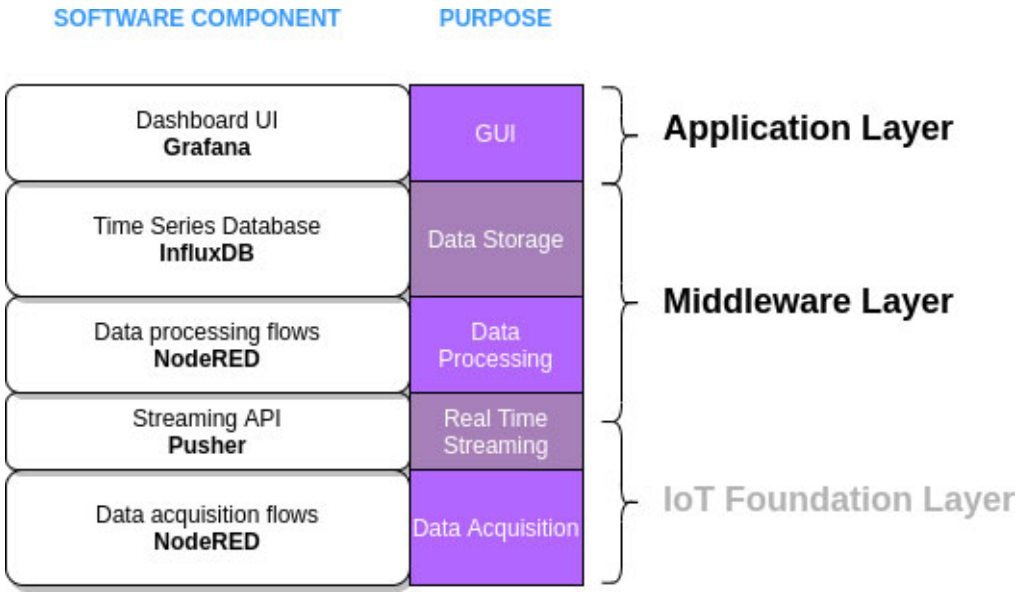


Figure 5.1: *The whole application stack*

Be aware that NodeRED is split in two blocks:

- The **data acquisition flows**: Namely Sense Hat and Global context—read sensor data and stream them to Pusher.
- The **data processing flows**: Basic statistics and Store aggregated data—read data from Pusher, perform statistics calculations on them, and finally store the processed data in the database.

Therefore, when introducing the IoT hardware, the first set of flows will run in a NodeRED instance inside the Raspberry Pi, while the second set will be hosted in a remote server. This distribution is visually depicted in the following figure:

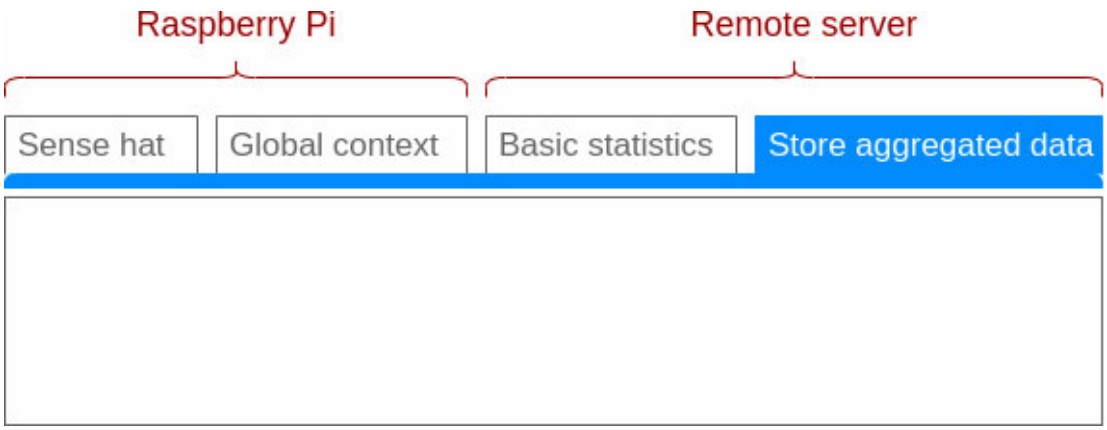


Figure 5.2: *The software stack*

The tab colored in blue is the one we will cover in this chapter. At this stage of the application development, both sets will be together in a NodeRED instance running in your laptop.

Complimentarily with the concept of the software stack, if changing the perspective to the point of view of the data, we say that there is a workflow described by means of a pipeline, as shown in *figure 5.1*: each software layer takes the data, does something on them, and delivers to the next layer upwards. Hence, the data flows from the bottom to the top of the software stack.

In the last part of the chapter we will describe the uppermost layer, i.e., the application layer, where we will visually represent a stream with Grafana, an application that provides dashboards for visualizing data. As explained in *Chapter 1, Introduction to IoT applications and their*

Software Architecture, this layer is stacked on top of the middleware layer.

The next section introduces InfluxDB and guides you to set up this new component of the software stack.

Setting up an InfluxDB database

InfluxDB is a **Not Only SQL** (**NoSQL**) database created by the company Influxdata. SQL is the acronym for Structured Query Language, the one used by relational databases like MySQL, SQLite, or PostgreSQL.

A relational database is composed of a series of tables that are linked between them by related columns. For example, you can have a table for the users of an application and another table listing the organizations those users belong to. The last table specifies the country where each organization is located. Therefore, if you want to build a table that tells you how many users there are in each country, you will have to combine both tables (users and organizations) to provide that information. The structure that describes the relation between tables is part of what is known as the database schema.

This brief description about relational databases serves the purpose of explaining the concept of non-relational databases—also known as NoSQL—the definition for which is not necessarily relational, even not requiring a database schema. Hence, NoSQL is a more general concept than SQL, and it responds to the nature of those databases that were conceived to overcome the limitations of the relational ones. This need was the consequence of new applications requesting to manage real-time streams, as well as huge volumes of data. Therefore, their development is closely related to the concept of Big Data.

InfluxDB is open source and, as in other cases of this kind of software when it is built by companies, it is also offered as a service in the cloud. Hence, it has the double advantage of being freely available for the community, as well as offered as a paid service when clients need to scale and avoid dealing with administration and maintenance.

Why don’t we opt for a more common NoSQL database like MongoDB? For sure it would also work, but in such a case, we would additionally

need to build the specific code to include a timestamp with every sensor measurement, something that InfluxDB performs automatically by including the system timestamp with every write operation.

Furthermore, querying the database to obtain measurements at a certain instant, or producing mean and standard deviation for a given period, would also require a specific code to be developed on your side. InfluxDB provides these and many other out-of-the-box time-dependent features, so that you can quickly get started. A time series database like InfluxDB natively offers such wrappers so that common operations in the database are simple and fast to carry out.

First we will explain the installation process of InfluxDB, and later we will start interacting using the basic mechanism of the command line.

Installation

Setting up InfluxDB is a straightforward process that we cover in the following step-by-step:

1. Add the repository of your Ubuntu distro:

```
$ echo "deb https://repos.influxdata.com/ubuntu $(lsb_release
-sc) stable" | sudo tee /etc/apt/sources.list.d/influxdb.list
```

> **Tip:** You can check the characteristics of your Ubuntu version with the following command:
>
> ```
> $ cat /etc/lsb-release
> DISTRIB_ID=Ubuntu
> DISTRIB_RELEASE=16.04
> DISTRIB_CODENAME=xenial
> DISTRIB_DESCRIPTION="Ubuntu 16.04.6 LTS"
> ```
>
> If you source this file, the environment variables will be created (not needed for this installation):
>
> ```
> $ source /etc/lsb-release
> ```

2. Import GPG key to check that the source repository can be trusted:

```
$ sudo curl -sL https://repos.influxdata.com/influxdb.key |
sudo apt-key add -
```

3. Update the list of repositories and install InfluxDB:

```
$ sudo apt-get update
$ sudo apt-get install influxdb
```

4. Finally start the service, and enable it to start on every boot up:

```
$ sudo systemctl enable --now influxdb
$ sudo systemctl start influxdb
```

5. Check that the service has a proper status:

```
$ systemctl status influxdb
```

At this point, you are ready to start interacting with the database.

Connecting from the command line

Open an InfluxDB session by typing the following command:

```
$ influx
Connected to http://localhost:8086 version 1.8.1
InfluxDB shell version: 1.8.1
>
```

This opens a new session that can be identified by the symbol '>'. The first thing you can do is to look at any existing database by typing:

```
> SHOW DATABASES
name: databases
name
----
_internal
```

Tip: InfluxDB query language is not case sensitive, so you can write commands and keywords in uppercase or lowercase. Hence, we could also have written > show databases. It is common to write them in uppercase, while measurements and field names are written in lowercase.

You can see that there is only one database, i.e., _internal that collects system information of the host machine where InfluxDB is installed.

InfluxDB authentication

The most basic level of security is to use an admin user with a custom password, especially if making requests over http. Although most of the work that follows will be done without authentication to avoid one more layer of complexity, it is convenient that you know how to add this basic security function.

To enable authentication, modify in the configuration file `influxdb.conf` the `[http]` section as follows:

```
$ sudo nano /etc/influxdb/influxdb.conf
[http]
  enabled = true
  auth-enabled = true
```

Then, restart `influxdb` service so that the change takes effect:

```
$ sudo systemctl restart influxdb
```

Next, create a user with an authentication password (let's take admin as 'user' and 'raspberry' as password for the example below). This is something that we make from the command line, making a http request with the `curl` command:

```
$ curl -XPOST "http://localhost:8086/query" --data-urlencode
"q=CREATE USER admin WITH PASSWORD 'raspberry' WITH ALL
PRIVILEGES"
```

Now, whenever you need to run any `influxdb` command on the terminal, specify `-username` and `-password` options as follows:

```
$ influx -username admin -password 'raspberry'
Connected to http://localhost:8086 version 1.8.1
InfluxDB shell version: 1.8.1
>
```

Be aware that the password has to be wrapped with single quotation marks. If you wish to interact using the `http` request mechanism, add the authentication to `curl` with the option `-u` to specify the username and password. For example, to run the InfluxDB query `SHOW DATABASES` do like this:

```
$ curl -G http://localhost:8086/query -u admin:raspberry --data-
urlencode "q=SHOW DATABASES"
{"results":[{"statement_id":0,"series":
[{"name":"databases","columns":["name"],"values":
[["_internal"]]}]}]}
```

By default, InfluxDB service listens on all network interfaces at port `8086`.

InfluxDB query language in brief

If you are familiar with **Structure Query Language** (**SQL**) databases like MySQL or PostgreSQL, you will find that the syntax InfluxDB queries are pretty similar.

Connect with the `-precision rfc3339` option to see the timestamps in a human readable format (YYYY-MM-DDTHH:MM:SS.nnnnnnnnnZ):

```
$ influx -precision rfc3339 -username admin -password 'raspberry'
```

Note: The option -precision rfc3339 makes timestamps visualized in UTC time ISO 8601 format, i.e., 2020-08-14T07:35:30Z, where the Z indicates that the reference is UTC (Coordinated Universal Time). The advantage of this format is that it is time-zone independent, it is the same value for everybody all over the world. Finally, be aware that InfluxDB natively stores the time in nanoseconds. This feature allows covering situations in which high time accuracy is requested in the measurements.

Once connected, find the databases that exist:

```
> SHOW DATABASES
name: databases
```

```
name
_internal
```

There is only one called `_internal`, that contains system-level statistics information. Switch to that database as follows:

```
> USE _internal
```

An InfluxDB database contains measurements, a concept similar to tables in general purpose databases like MySQL. By calling it measurements, we make them more significant with respect to the kind of data it contains, i.e., measurements of a variable. Let's find what time series we have in the `_internal` database:

```
> SHOW MEASUREMENTS
name: measurements
name
cq
database
httpd
queryExecutor
runtime
shard
subscriber
tsm1_cache
tsm1_engine
tsm1_filestore
tsm1_wal
write
```

You can find that there are 12 series, each one with the given name. So, the way to refer to data series in InfluxDB is the concept of measurement.

One important concept in time-oriented databases is how long information is kept. In InfluxDB, this is specified as the retention policy, and can be found for a specific database with the following command:

```
> SHOW RETENTION POLICIES ON _internal
name    duration shardGroupDuration replicaN default
```

```
----    -------- ----------------- -------- -------
monitor 168h0m0s 24h0m0s            1        true
```

The `duration` field tells us how long our data is kept stored, i.e., 168 hours, in what InfluxDB calls shards. Data points older than such a time span are deleted. In this way, we avoid the database size growing indefinitely, while keeping track of the detailed historic records that we really need.

Given that we have installed InfluxDB on a single machine, we only have one node and the concept of a cluster does not apply. In any case, it is worth using this discussion about the retention policy to briefly explain how it would work.

A cluster of InfluxDB nodes organizes the data in shard groups, whose way of working is as follows:

- Every shard contains temporal blocks of data. Hence, the set of shards in a group contains the full history of our time series.
- Every shard stores a partial set of the data. For example, if there are two shards, it means that each one will contain half the data points, and each shard will keep those records for the last 168 hours.

The *shard group duration* refers to the time a shard group lives. Although the shard group expires (24 hours in the example above), if the duration of the individual shards is longer (168 h), it means that they will stay alive for 168-24=144 hours more. So, InfluxDB will automatically create a new shard group and will include in it the living shards.

The other concept in the retention policy relevant to a cluster is the replication number, which refers to the copies that are made of every data point. Hence, the replication factor applies to the individual shards, and if greater than one, it will mean that every shard will be replicated in another node of the cluster.

Following the exploration of the measurements, to obtain their characteristics, we can write the following command:

```
> SHOW SERIES ON _internal
```

```
key
---
cq,hostname=laptop
database,database=_internal,hostname=laptop
httpd,bind=:8086,hostname=laptop
queryExecutor,hostname=laptop
runtime,hostname=laptop
shard,database=_internal,engine=tsm1,hostname=laptop,id=1,indexTy
pe=inmem,path=/var/lib/influxdb/data/_internal/monitor/1,retentio
nPolicy=monitor,walPath=/var/lib/influxdb/wal/_internal/monitor/1
subscriber,hostname=laptop
tsm1_cache,database=_internal,engine=tsm1,hostname=laptop,id=1,in
dexType=inmem,path=/var/lib/influxdb/data/_internal/monitor/1,ret
entionPolicy=monitor,walPath=/var/lib/influxdb/wal/_internal/moni
tor/1
tsm1_engine,database=_internal,engine=tsm1,hostname=laptop,id=1,i
ndexType=inmem,path=/var/lib/influxdb/data/_internal/monitor/1,re
tentionPolicy=monitor,walPath=/var/lib/influxdb/wal/_internal/mon
itor/1
tsm1_filestore,database=_internal,engine=tsm1,hostname=laptop,id=
1,indexType=inmem,path=/var/lib/influxdb/data/_internal/monitor/1
,retentionPolicy=monitor,walPath=/var/lib/influxdb/wal/_internal/
monitor/1
tsm1_wal,database=_internal,engine=tsm1,hostname=laptop,id=1,inde
xType=inmem,path=/var/lib/influxdb/data/_internal/monitor/1,reten
tionPolicy=monitor,walPath=/var/lib/influxdb/wal/_internal/monito
r/1
write,hostname=laptop
```

Every line specifies a single measurement. Since we have previously selected the `_internal` database, we could have omitted its name in the command.

The first word in bold letters is the name, it identifies each measurement. The rest of each line provides the features of each time series.

Let's look at some sample content from one of the measurements. The command is equivalent to the one we would use in SQL:

```
> SELECT HeapAlloc FROM runtime LIMIT 5
name: runtime
time                  HeapAlloc
----                  ---------
2020-08-14T07:35:30Z  3999488
2020-08-14T07:35:40Z  3516064
2020-08-14T07:35:50Z  3925456
2020-08-14T07:36:00Z  4180936
2020-08-14T07:36:10Z  4434304
```

We are listing the field `HeapAlloc` from the last five points of the measurement runtime.

Note: QL queries mandatorily end with a semicolon. In InfluxDB, this feature is optional, i.e., you can use the semicolon or not. If not used, the query is assumed to end with the line.

At this point, we have a basic understanding of InfluxDB to proceed to set up a database for hosting our measurements of the environmental conditions.

Setting up the environment database

We learned above how to set up authentication, but for the purpose of simplicity we are going to skip this feature from now on.

The way to remove the authentication option is to undo some of the modifications we performed in the InfluxDB configuration file, setting the `auth-enabled` option to false:

```
$ sudo nano /etc/influxdb/influxdb.conf
[http]
  enabled = true
  auth-enabled = false
```

Be aware that we keep http enabled anyway, i.e., `enabled = true`, since this allows us to perform queries not only from the command line, but also via http requests.

Then, restart `influxdb` service so that the change takes effect:

```
$ sudo systemctl restart influxdb
```

Now, connect with as we did before, but skipping the authentication parameters:

```
$ influx -precision rfc3339
```

Then, create the `environment` database:

```
> CREATE DATABASE environment
```

And now, let's put some data inside. From the code of this chapter, change to the folder `./influxDB`, where you have the sample data file `temperature.txt`:

```
$ cd ~/book_hands-on-iot/chapter5/influxDB
```

Its contents are just two data points:

```
temperatures,location=HOME
room="KITCHEN",instant=28.5,average=28.45 1597397160
temperatures,location=HOME
room="KITCHEN",instant=28.4,average=28.42 1597397100
```

The first parameter in each line, `temperatures`, refers to the measurement.

It follows a set of the contextual features, named tags (`location` and `room` in our example), that are separated by spaces. If there are several, they constitute a tag set.

Finally—separated by colons—we have the fields. They contain values on a point and constitute a field set.

Note: Fields store your data, while tags store the metadata, i.e., information that provides context to your data. As in SQL, tags are indexed columns that allow performing data

searches. Finally, be aware that fields are treated as numeric values, while tags are processed as strings.

The last number in each line is optional, and it refers to the timestamp:

- If you specify a value, it will correspond to UTC in seconds taking as reference 1 January 1970, i.e., UNIX time.
- If you do not write anything, InfluxDB will automatically insert the current timestamp.

Tip: At this point, you can appreciate what a convenient feature this one is. When you push real-time sensor data to InfluxDB, you will not have to add the timestamp to the data point previously; InfluxDB will do it for you. Hence, the only point you have to be careful of is that the system time of your machine is correct. This could be an issue with Raspberry Pi, which does not have a built-in clock. It relies on the Internet connection to keep the system on time.

You can import the file from a terminal using the `influx` command with the following options:

```
$ influx -import -path=temperature.txt -database=environment -
precision=s
```

Here, you specify the file and the target database. The last parameter, `-precision=s`, tells that the timestamps are specified in seconds. Be aware that InfluxDB natively stores timestamps in nanoseconds, so it is crucial to specify the unit of time of your import.

You can check the official documentation of the `influx` command at **https://docs.**

```
influxdata.com/influxdb/v1.8/tools/influx-cli/
```

Note: If authentication were to be kept, just add to the command the username and password options:$ influx -username admin -password 'raspberry' \

```
-import -path=temperature.txt -database=environment -precision=s
```

Now, log into InfluxDB from the command line and check the result:

```
$ influx
> USE environment
Using database environment
> SHOW MEASUREMENTS
name
----
temperatures
> SELECT * FROM temperatures
name: temperatures
time                average instant location room
----                ------- ------- -------- ----
1597397100000000000 28.42   28.4    HOME     KITCHEN
1597397160000000000 28.45   28.5    HOME     KITCHEN
```

Be aware that we did not specify the option `-precision rfc3339`. Hence, timestamps will be shown in nanoseconds. You can confirm how every timestamp has added 9 zeros.

Note: To remove the measurement, you can use the command DROP followed by the name of the measurement:> DROP MEASUREMENT temperatures

If you have deleted the measurement, recreate it by repeating the command above, because you will need it in the next section.

Now, we are ready to switch to our actual scenario, in order to learn how to push data points from NodeRED to the environment database we have created.

Storing time series of environmental conditions

The setup of InfluxDB nodes in NodeRED is pretty straightforward as we will show in the following paragraphs. First, you will learn to query the temperature data that we imported in the previous section. Then, we will write the data points.

Reading an InfluxDB database from NodeRED

On the NodeRED editor, add a blank canvas to sketch the flows regarding the basic read and write operations for the database.

The most basic version consists of placing an inject node—with a timestamp payload—a debug node to visualize the output, and an Influxdb in node between them as shown in the following figure:

Figure 5.3: *Query InfluxDB database from NodeRED: Option 1*

When you first click on the `Influxdb in` node, you will have to add a new connection by clicking on the edit icon to the right of the `Server` field. Then, a new form will be open as shown in *figure 5.4* below, where you have to specify host, port, and database name:

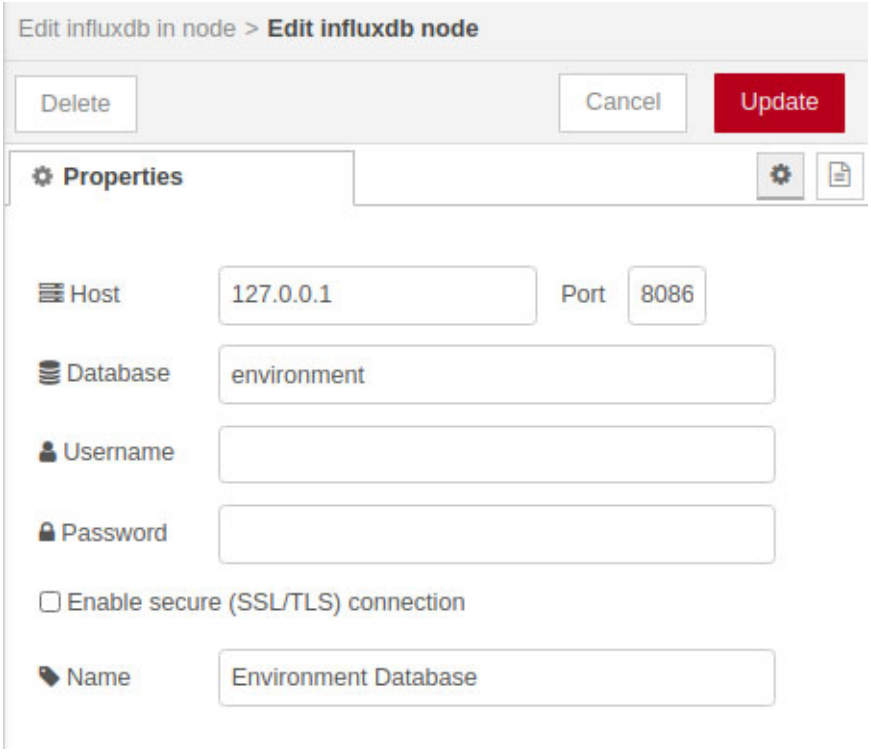


Figure 5.4: *Adding an InfluxDB database connection*

Since you installed InfluxDB on the same machine where NodeRED is running, the host parameter is localhost or `127.0.0.1` (they are equivalent), the port is the default `8086`, and the database is

environment, as created in the previous section. Given that you configured InfluxDB without authentication, you can leave username and password empty. Finally, the **`Name`** field refers to the designation you want to give to this connection within the scope of NodeRED.

Tip: To make sure what the host and port parameters are, you can find them when opening a terminal for the InfluxDB command:

```
$ influx
Connected to http://localhost:8086 version 1.8.1
InfluxDB shell version: 1.8.1
```

Look that the string connection is shown putting together both parameters, i.e. `http://localhost:8086`

After configuring the server and closing its form, you will be back in the **`Influxdb in`** node form. Then, you can fill in the query in the corresponding field (**`SELECT * FROM temperatures`**) as shown in the following figure:

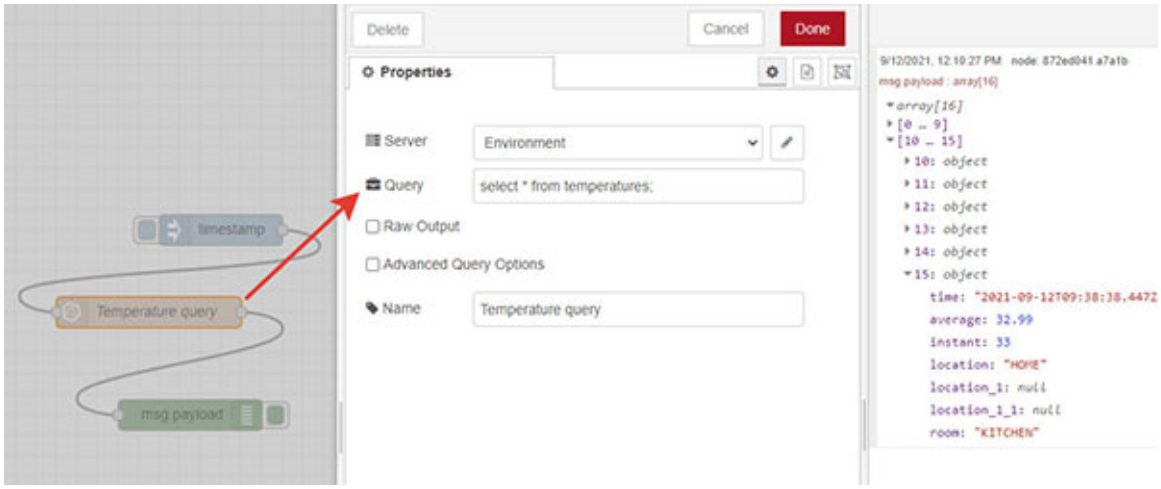


Figure 5.5: *Building the query for the Influxdb in node*

Connect the three nodes, deploy the flow and click on the inject node to see the output in the debug tab. As shown in the figure above, there will be two JSON objects, one per each line in the file **`temperature.txt`**.

Alternatively, you can build the query in a function node and pass it to Influxdb as the attribute query of the **`msg`** object, i.e., **`msg.query`** as

shown in the following figure

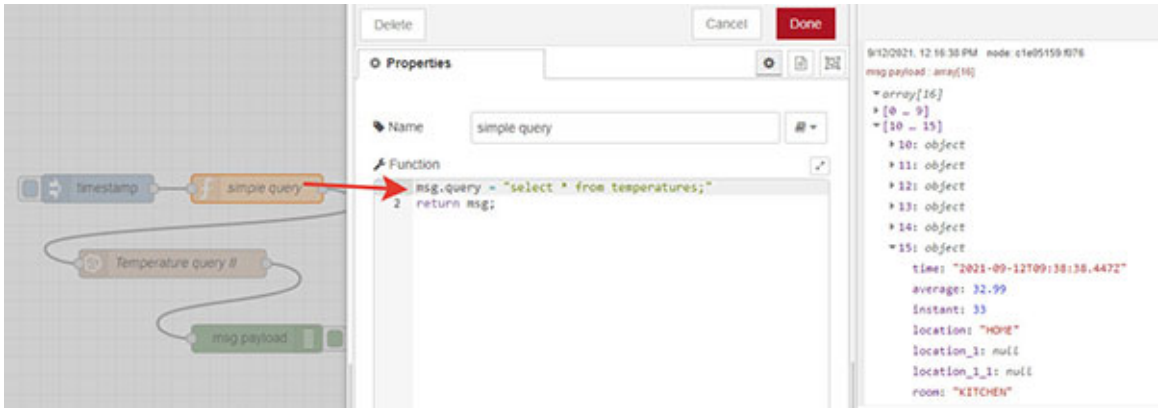


Figure 5.6: *Building externally the query to InfluxDB: Option 2*

The next exercise will consist of writing a data point from the flow itself using the **`Influx out`** node.

Writing data points from NodeRED

First, let's learn how to insert one data point from the InfluxDB terminal. To do so, open a terminal and connect as usual, then select the environment database:

```
$ influx -precision rfc3339
> USE environment
Using database environment
```

Perform the write operation prepending the **`INSERT`** command and copying any of the lines in the file **`temperature.txt`**.

```
> INSERT temperatures,location=HOME
room="KITCHEN",instant=30.9,average=29.55
```

Since you did not specify a timestamp, InfluxDB will use the system time. If you perform a **`SELECT`** query, you should find that the data point has been written in the temperature's measurement (highlighted in bold letter below):

```
> SELECT * FROM temperatures
name: temperatures
time                           average instant location room
```

```
----                          -------  -------  --------  ----
2020-08-14T09:25:00Z          28.42    28.4     HOME      KITCHEN
2020-08-14T09:26:00Z          28.45    28.5     HOME      KITCHEN
2020-08-14T11:44:44.974346601Z 31.02   32.5     HOME      KITCHEN
```

In order to perform the same, write operation from NodeRED (refer *figure 5.7*), use an InfluxDB out node and prepend a function node where you will add the `INSERT` command:

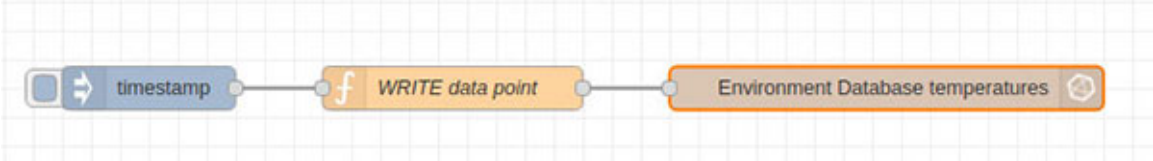


Figure 5.7: *Write a data point in InfluxDB from NodeRED*

First, configure the Influxdb out node specifying the Server connection (Environment Database in *figure 5.8*) and the name of the `Measurement`, i.e., `temperatures`:

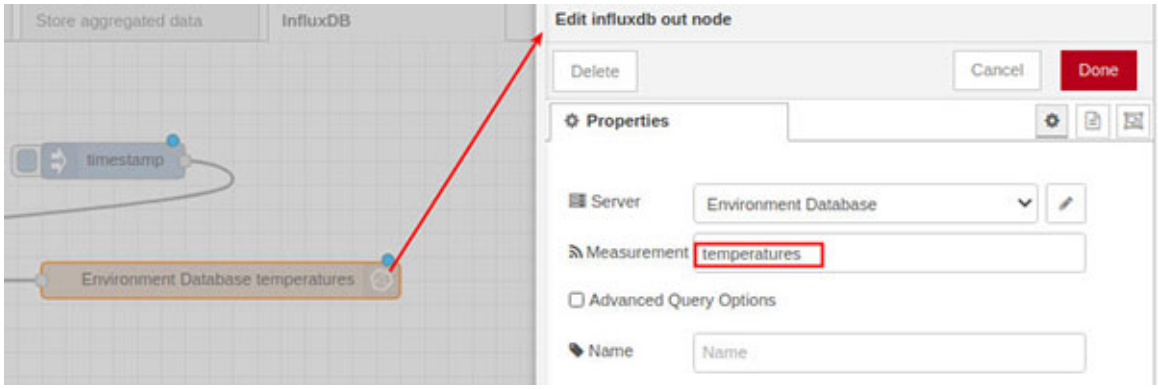


Figure 5.8: *Store sensor data to InfluxDB from NodeRED*

Then, write the attributes of the data point in the `msg.payload` of the `Function` node as shown in the following figure:

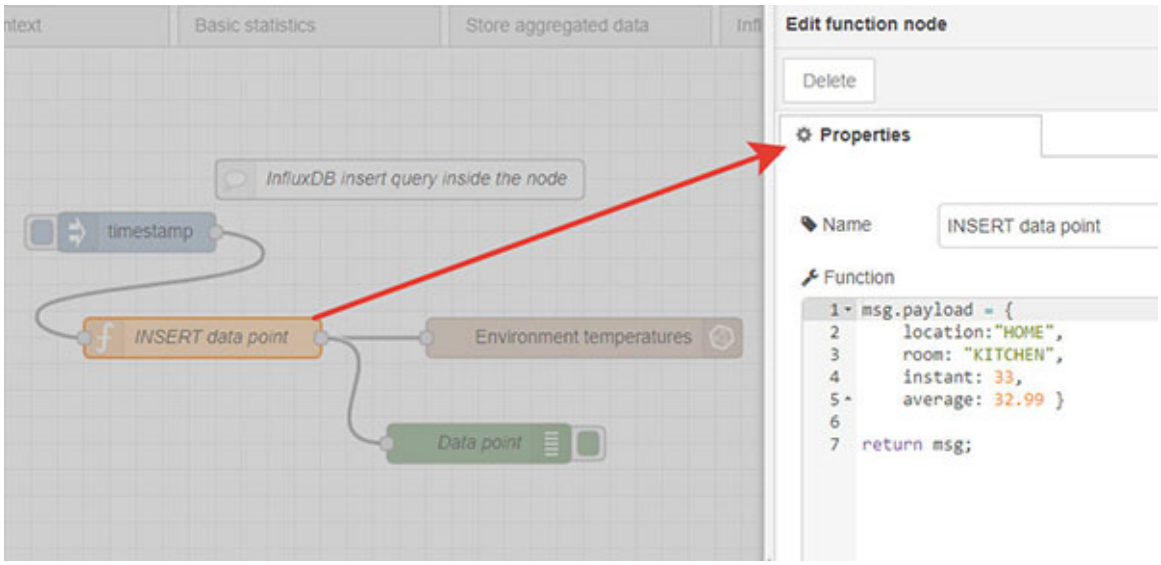


Figure 5.9: *Payload preparation to store sensor data in InfluxDB*

Be aware that `string` attributes will be treated as `InfluxDB` tags, while numeric values will be considered fields. Hence, if you want to add a tag that reads like a number, you should wrap it between double quotes so that InfluxDB interprets it.

Finally, check the result as usual:

```
> select * from temperatures
name: temperatures
time                           average instant location   room
----                           ------- ------- --------   ----
2020-08-14T09:25:00Z           28.42   28.4    HOME       KITCHEN
2020-08-14T09:26:00Z           28.45   28.5    HOME       KITCHEN
2020-08-14T11:44:44.974346601Z 31.02   32.5    HOME       KITCHEN
2020-08-14T12:07:18.260185333Z 32.99   33      HOME       KITCHEN
```

Figure 5.10: *List points of temperature's measurement*

The new data points are added at the end of the table. So, the last row corresponds to the data point you have just inserted. You can also perform a check by clicking on the `Inject` node of the flow we created in the previous section.

Up to this point, we have learned how to read and write data points from NodeRED. In the next section, we will apply these methods to create the measurement for the temperature reading of the Sense Hat simulator.

Storing processed data of environmental conditions

This section explains the storage functionality for our application, for which we performed a preliminary version using a JSON static file that stores data points in plain text. This was covered in the section *Storing aggregated data for later processing* of the previous *Chapter 4, Real time data processing with NodeRED*.

Now that we have acquired the background of how to work with a time series database, we can replace the file storage for the robust and professional database management that InfluxDB provides. Hence, that version of the flow tab Store aggregated data will be replaced by the content we develop in this section.

To start the Node RED instance, run the following commands:

```
$ cd ~/book_hands-on-iot/chapter6/app
$ npm run flow2-CH4
```

Note: Remember that you have to configure pusher out and pusher in nodes so that the data from the Sense Hat simulator can be read and finally stored in the database. This was covered in the section Setup Pusher nodes to point to your account and app of *Chapter 4 Real time data processing with NodeRED*.

Compare the NodeRED flow below with the one we showed in *figure 4.27* of the previous chapter (in any case it is shown again together with the new version for comparison purposes):

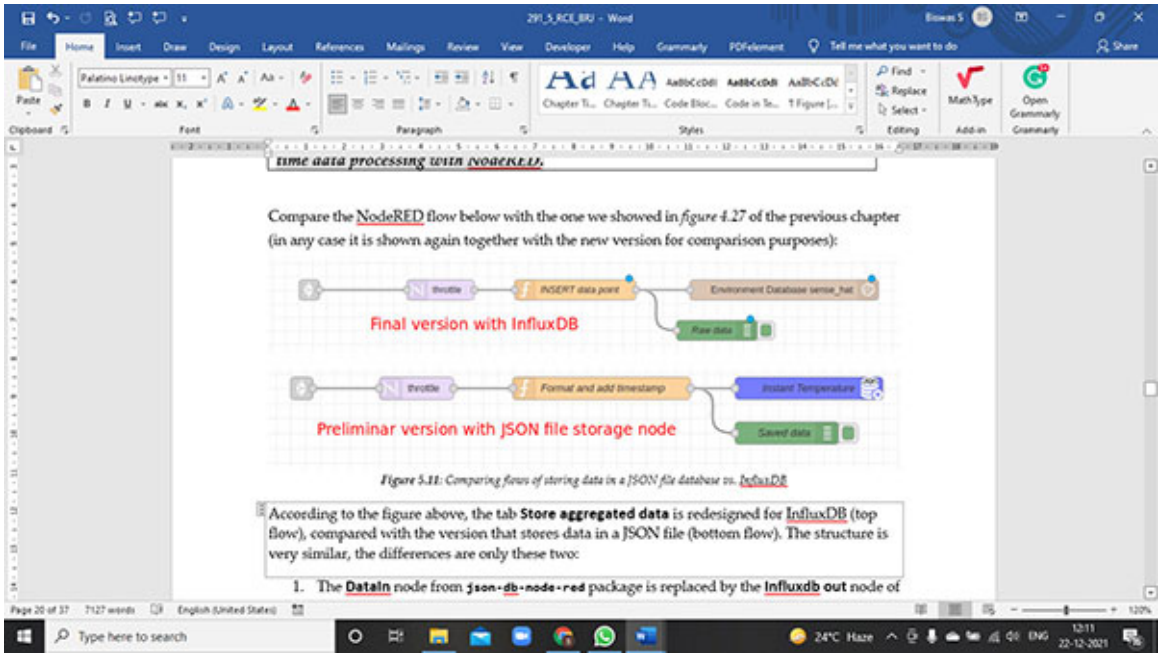


Figure 5.11: *Comparing flows of storing data in a JSON file database vs. InfluxDB*

According to the figure above, the tab `Store aggregated data` is redesigned for InfluxDB (top flow), compared with the version that stores data in a JSON file (bottom flow). The structure is very similar, the differences are only these two:

1. The `DataIn` node from `json-db-node-red` package is replaced by the `Influxdb out` node of `node-red-contrib-influxdb` package. This node is configured as explained in the previous section. You should specify a different InfluxDB measurement so that this data series does not mix with the one we used before. For example, we can name this new measurement `sense_hat` as shown in the following figure:

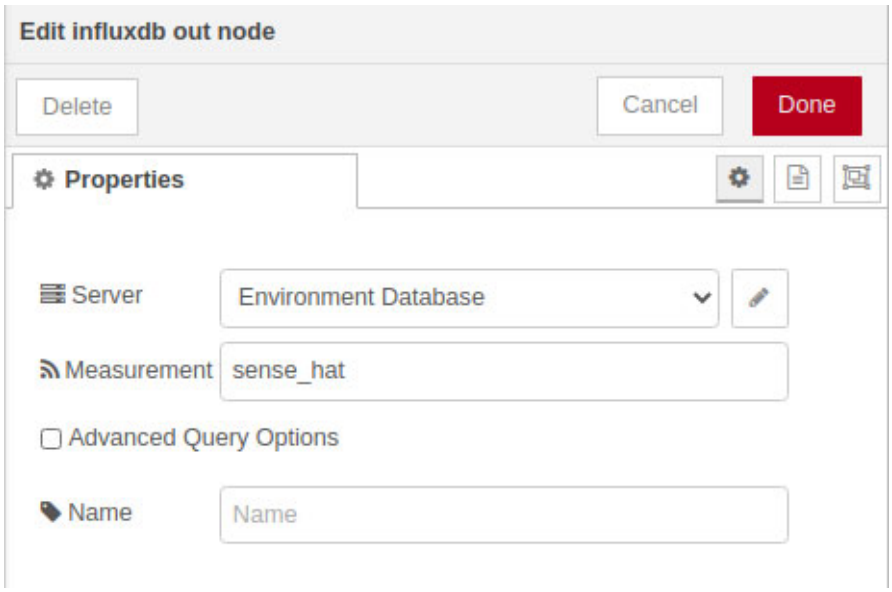


Figure 5.12: *Define the InfluxDB measurement where raw temperature data will be stored, i.e., sense_hat_avg_simulator*

2. The function node is simplified because you do not have to add the timestamp: InfluxDB makes it automatically for you whenever a new data point is inserted. The JavaScript content of this node is:

```
msg.payload = {location:"HANGAR", room: "Warehouse",
temperature: msg.payload}
return msg;
```

We can go one step forward and not only store the raw temperature data, but also a refined version like the one we built in the previous chapter in the section *Basic statistics of environmental conditions*.

To build it, starting from the top flow of the *figure 5.11* above, we only have to feed the real-time averaged temperature from the NodeRED tab Basic statistics instead of the raw measurements. The steps are as follows:

1. From the tab `Basic statistics`, take the output of the node designated as statistics over 10 (recap *figure 4.16* from the last chapter) and connect to a link out node (encircled in the next *figure 5.13*):

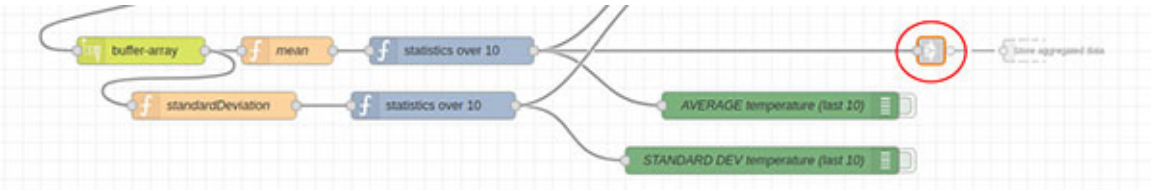


Figure 5.13: *Output from the Basic statistics tab*

2. Then, retrieve the feed of messages in the tab **`Store aggregated data`** by adding a link in node (encircled in the following *figure 5.14*) that you should configure so that it is connected to the link out node:

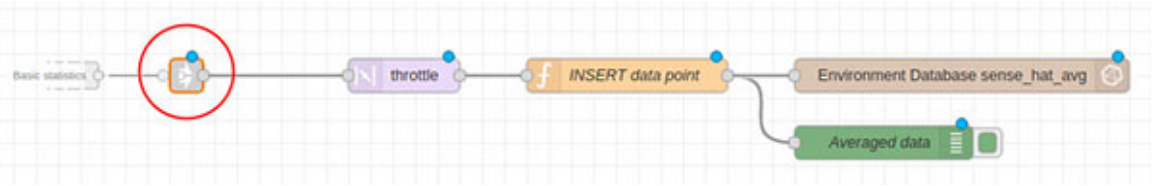


Figure 5.14: *Input to the Store aggregated data tab*

3. Finally, change the name of the InfluxDB measurement to **`sense_hat_avg`** so that it does not mix with that of the raw data (**`sense_hat`**) as shown in the following figure:

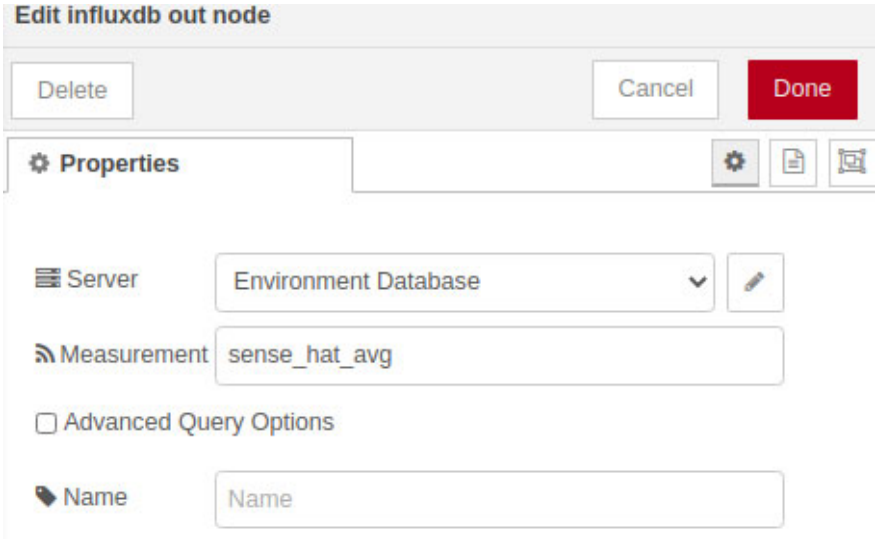


Figure 5.15: *Define the InfluxDB measurement where the average temperature data will be stored*

Deploy the NodeRED flows and check the result with the known **`influx`** command:

- First, for the raw temperature data:

```
$ influx -precision rfc3339
> use environment
Using database environment

> select * from sense_hat;
name: sense_hat
time                           location room      temperature
----                           -------- ----      -----------
2020-08-16T16:43:32.585415631Z HANGAR   Warehouse 20
2020-08-16T16:43:42.656727822Z HANGAR   Warehouse 20
2020-08-16T16:43:53.015880947Z HANGAR   Warehouse 20.8
2020-08-16T16:44:03.193781385Z HANGAR   Warehouse 21.4
2020-08-16T16:44:13.493146538Z HANGAR   Warehouse 21.4
2020-08-16T16:44:25.49479549Z  HANGAR   Warehouse 21.4
```

- Then, for the average temperature:

```
> select * from sense_hat_avg;
name: sense_hat_avg
time                           location room      temperature
----                           -------- ----      -----------
2020-08-16T16:43:32.586706905Z HANGAR   Warehouse 19.96
2020-08-16T16:43:42.661962169Z HANGAR   Warehouse 20
2020-08-16T16:43:53.016500689Z HANGAR   Warehouse 20.16
2020-08-16T16:44:03.19519329Z  HANGAR   Warehouse 20.8
2020-08-16T16:44:13.494176492Z HANGAR   Warehouse
21.339999999999996
2020-08-16T16:44:25.495434361Z HANGAR   Warehouse 21.4
```

With this section, we have achieved a crucial goal for our application, that is to make our data independent of the NodeRED application. Hence, from now on, we can access temperature measurements in real time without needing to be in front of the NodeRED editor, just by querying the InfluxDB database.

In the next section, we will set up Grafana to visualize real-time data in friendly dashboards.

Creating a visualization dashboard with Grafana

Grafana is an open source visualization and analytics software. It allows you to query, visualize, alert on, and explore the metrics of data stored in a database.

First, let's go through the installation steps of this tool.

Installation

Follow this procedure to get Grafana installed and running:

Step 1: Update system

As usual, proceed to update the package sources to install the packages in step 2:

```
$ sudo apt-get update
```

Step 2: Add Grafana APT repository

Install the required dependencies:

```
$ sudo apt-get install -y gnupg2 curl software-properties-common
```

It is possible that these packages are already installed, but do it anyway to make sure.

Then, add the Grafana `gpg` key which allows you to install signed packages:

```
$ curl https://packages.grafana.com/gpg.key | sudo apt-key add -
$ sudo add-apt-repository "deb
https://packages.grafana.com/oss/deb stable main"
```

Step 3: Install Grafana

Update to include the new sources, and install Grafana:

```
$ sudo apt-get update
$ sudo apt-get -y install grafana
```

Step 4: Enable the Grafana service

An Ubuntu service makes an application always running, even after a reboot. So, enable it for the installed Grafana:

```
$ sudo systemctl enable --now grafana-server
```

Finally, check that it is working:

```
$ systemctl status grafana-server
```

You should see an output in the command line that tells you that the Grafana service is active and running.

Setting up the Grafana observability platform

You can access the Grafana configuration through the file **grafana.ini**:

```
$ sudo nano /etc/grafana/grafana.ini
```

The main parameter you have to take into account is the 'http' that the application uses. It is 3000 by default. But if you have another application running in that port, there will be a collision. So, you should change it to a free one:

```
# The http port to use
;http_port = 3030
```

You should also need to update the same configuration line in file **/usr/share/grafana/conf/defaults.ini**

Finally, restart the service so that changes take effect:

```
$ sudo systemctl restart grafana-server
```

Now, visit the following URL in the browser: **http://localhost:3030**

The first time you access the URL, you will be presented a login window as shown in the following figure:

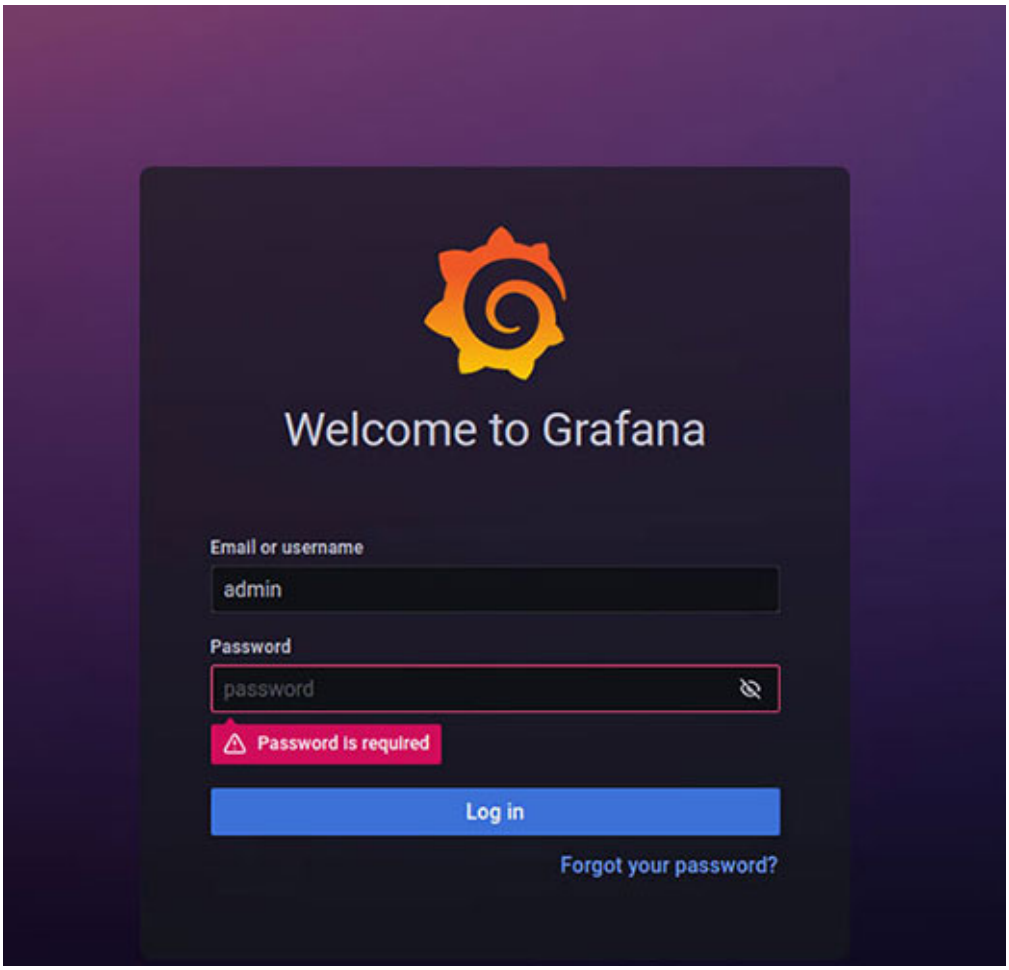


Figure 5.16: *Grafana login window (The first time you will be requested to create the admin password.)*

The default password is 'admin'. So, put it on the box and, on logging in, you will be requested to change it to a secret one. Choose the one you want, and after that, secure your access to the application in case you forget it. For that purpose, click on the profile icon and add your email address as shown in the following figure:

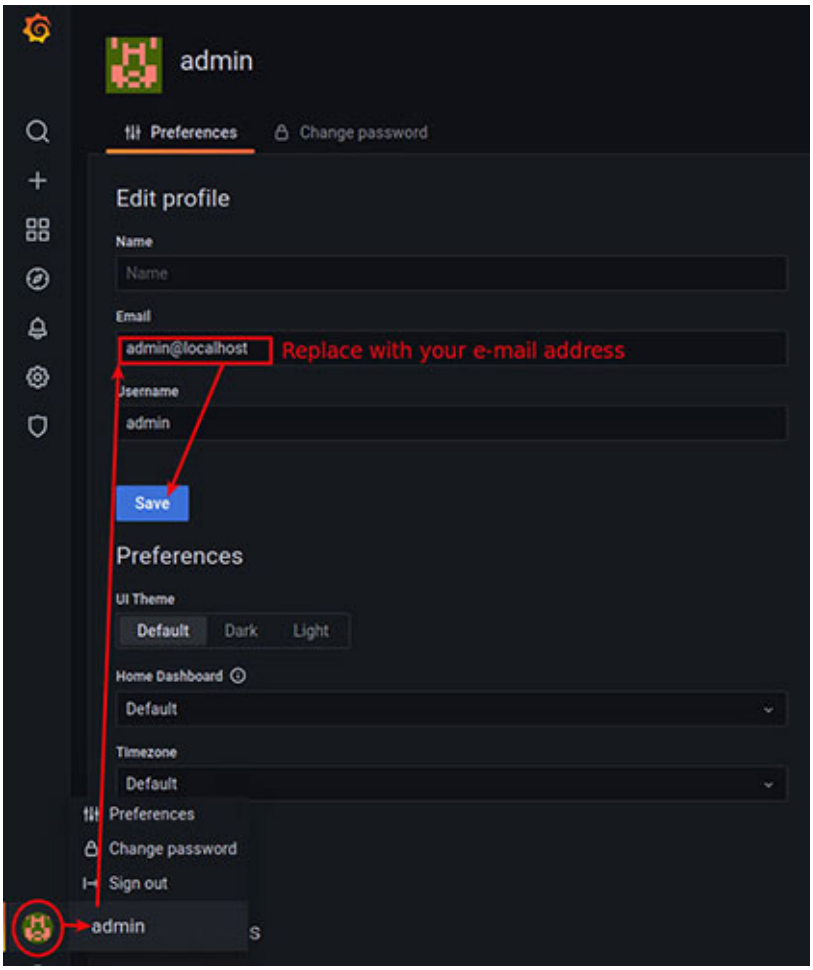


Figure 5.17: *Add your email address so that you can set a new password in case you forget it*

At this point, we are ready to create our first dashboard.

Creation of graph for the Sense Hat simulator

To create the graph of the temperature coming from the Sense Hat simulator, follow the next steps:

1. Go to `Data sources` as shown in the following figure:

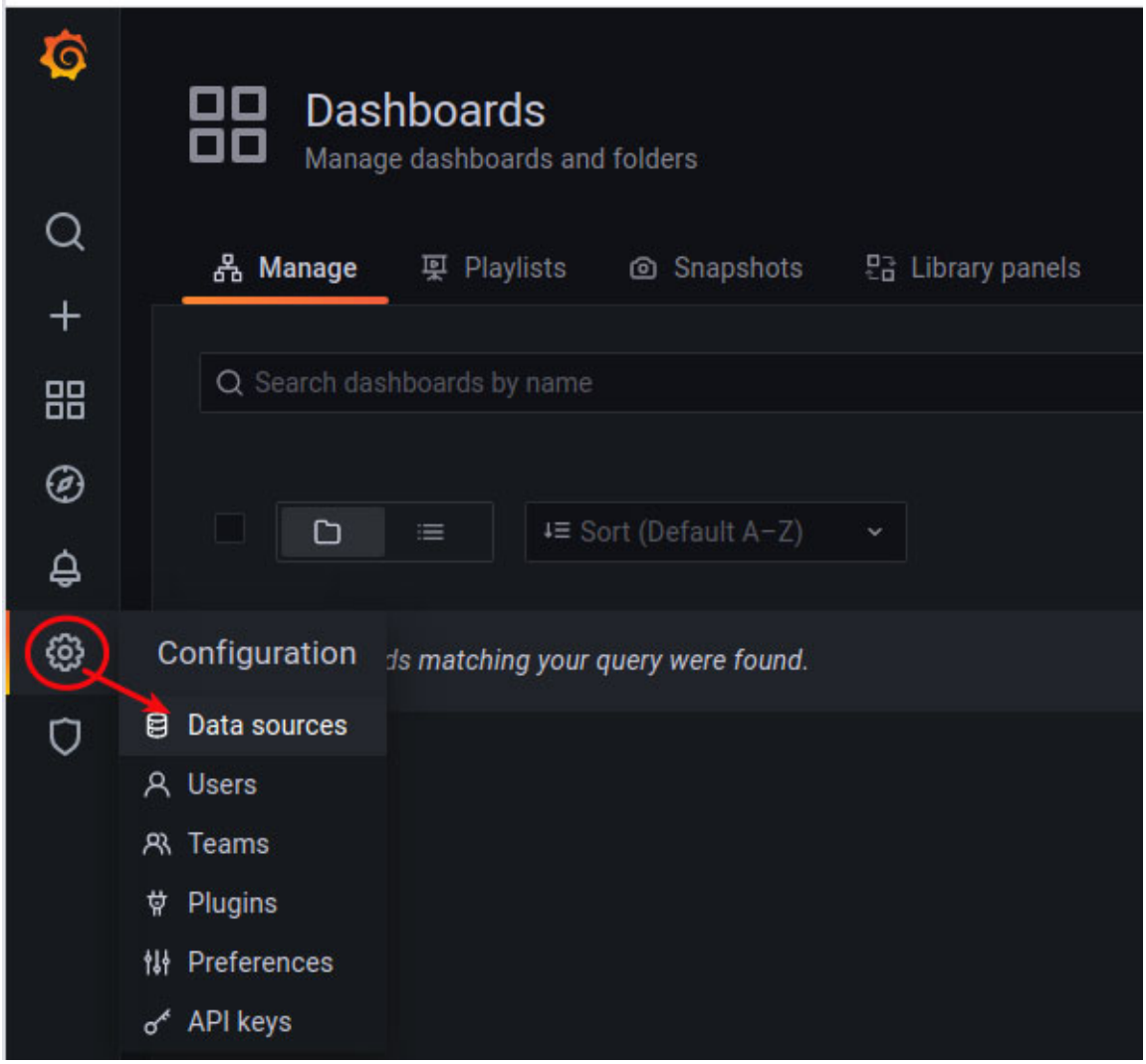


Figure 5.18: *Access data sources in settings menu*

2. Click on `Add data source` as shown in the following figure:

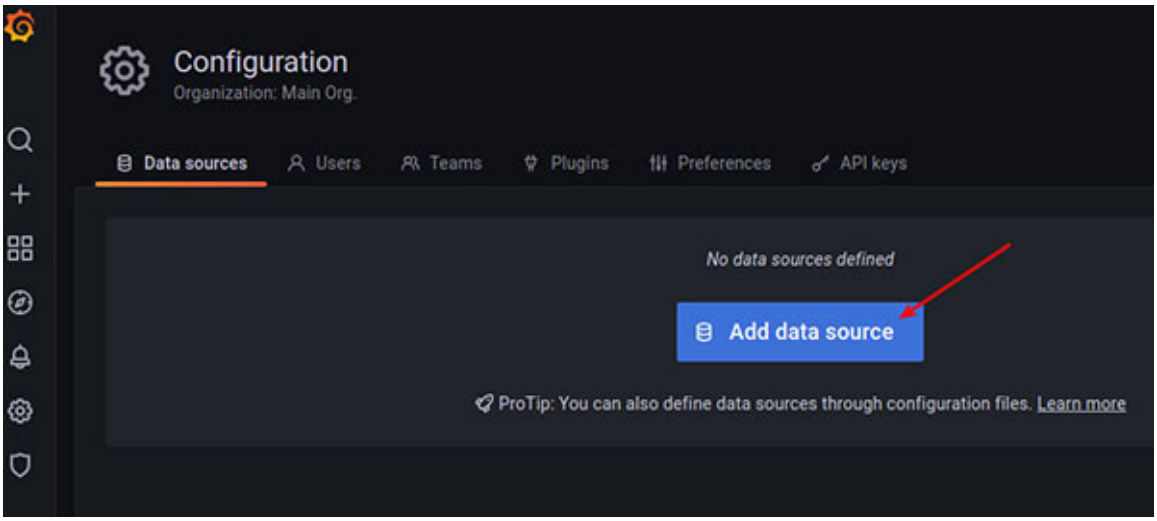


Figure 5.19: *Access the list of plugins to add a new source*

3. Select `InfluxDB` plugin from the list you will be using as shown in the following figure. Probably, it is faster if you type it in the search box:

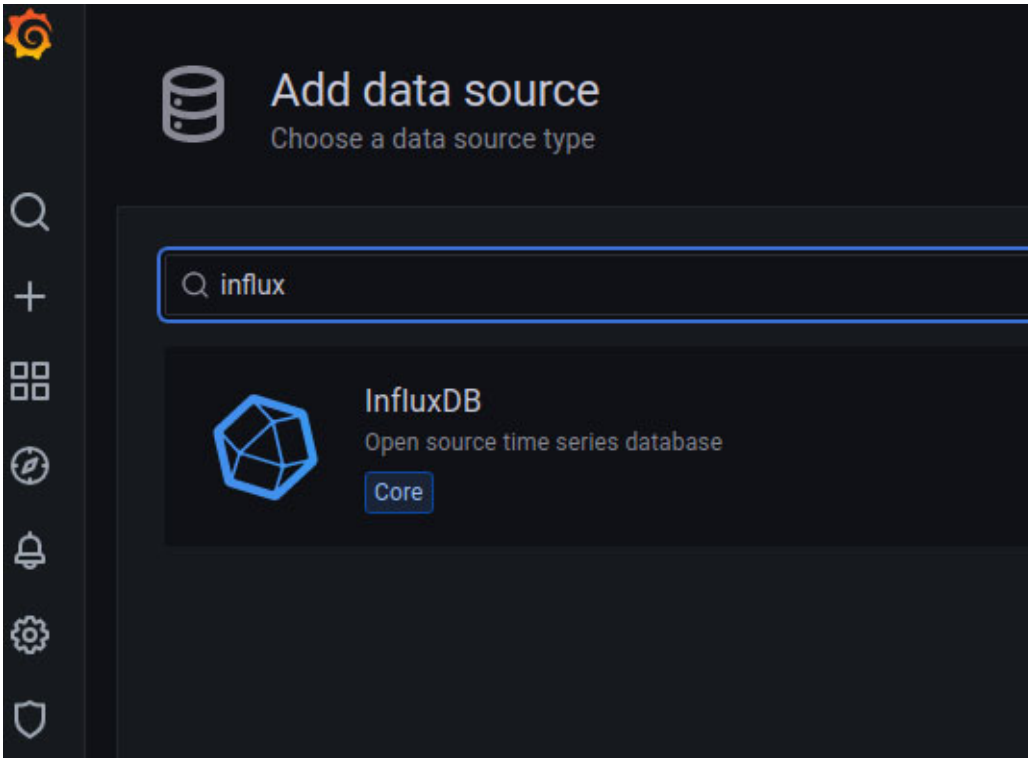


Figure 5.20: *Select InfluxDB data source*

4. Then, you will be redirected to the configuration window, where you will have to provide the URL of the InfluxDB server, i.e.,

`http://<IP_ADDRESS>:8086`. Since you have installed InfluxDB on the same machine, you could simply type in `http://localhost:8086`. But in any case, it is preferable to use the machine IP real, because otherwise, Grafana will not be able to find the database server if accessing the application from another PC in your local network.

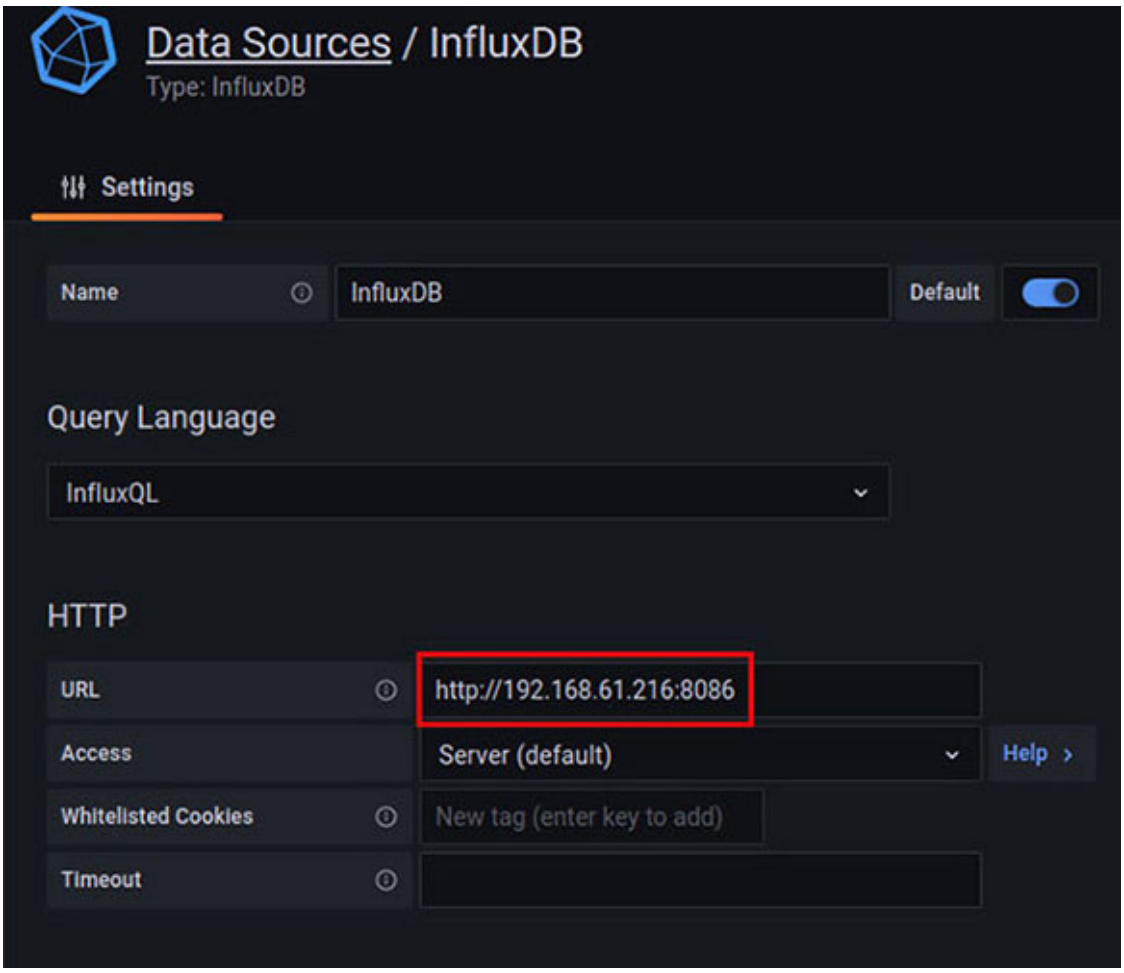


Figure 5.21: *Provide the URL of the InfluxDB server*

You can find your machine IP with the following command:

```
ubuntu@laptop:~$ ifconfig | grep "inet addr"
inet addr:127.0.0.1 Mask:255.0.0.0
inet addr:192.168.61.216 Bcast:192.168.61.255
Mask:255.255.255.0
```

127.0.0.1 refers to localhost, so the public IP is the other one, i.e. 192.168.61.216

5. After that, provide the name of the database where the Sense Hat data is being stored, i.e., environment, created as per paragraph. We set up the environment database above in this chapter (refer to *figure 5.22*). It was created for free access, so you do not have to set a user and password.

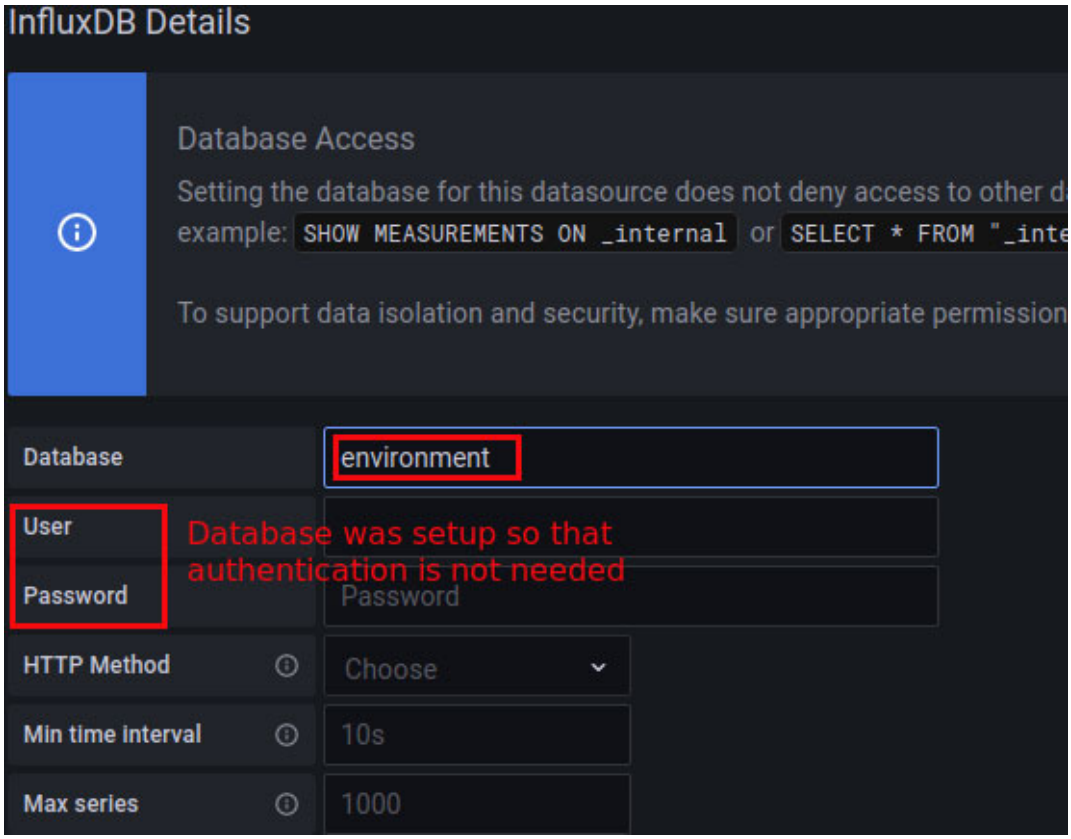


Figure 5.22: *Type in the database name*

6. Click on `Save & test` button (refer to *figure 5.23*), and you should receive the message `Data source is working`. If not the case, review the configuration window and check the parameters. If they are alright and still the connection does not work, you should check if InfluxDB is running. Remember that the command is:

```
$ sudo systemctl status influxdb
```

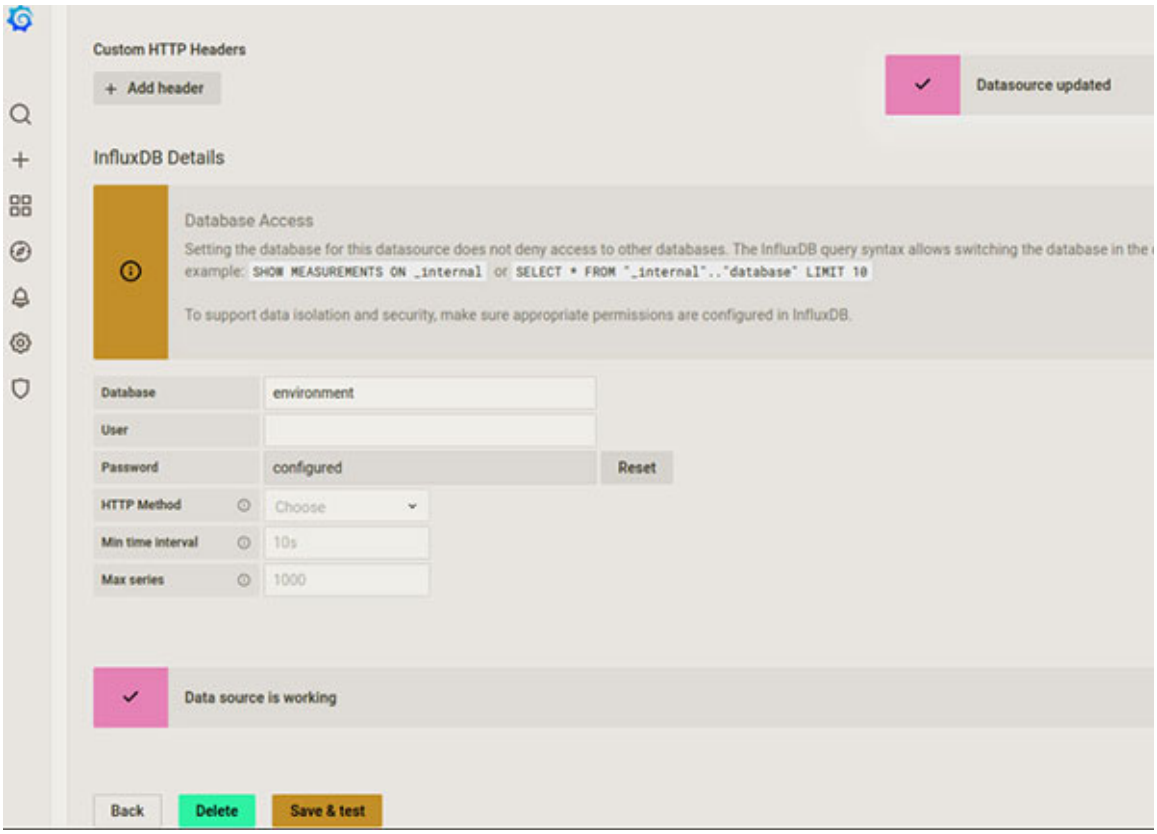


Figure 5.23: *Test and save the database connection*

7. Go to the `Dashboards` menu, and click on `Manage` as shown in the following figure:

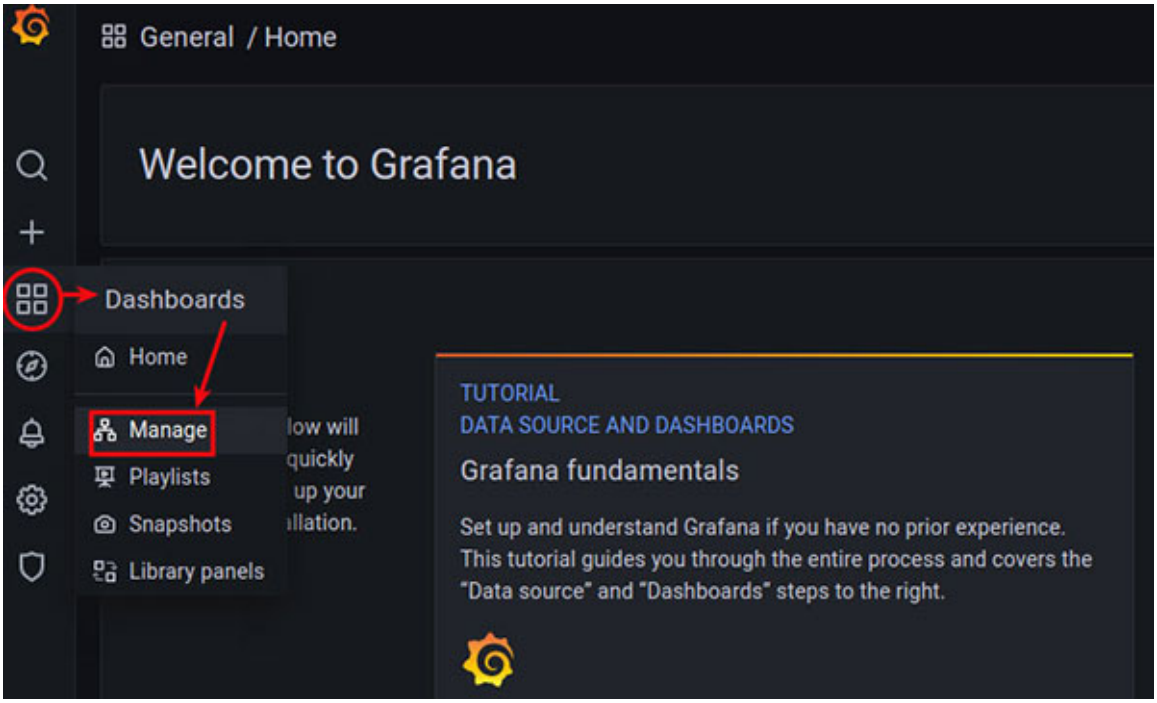


Figure 5.24: *Access the manage dashboards form*

8. Create a new dashboard by clicking on the button `New Dashboard`, as shown in the following figure:

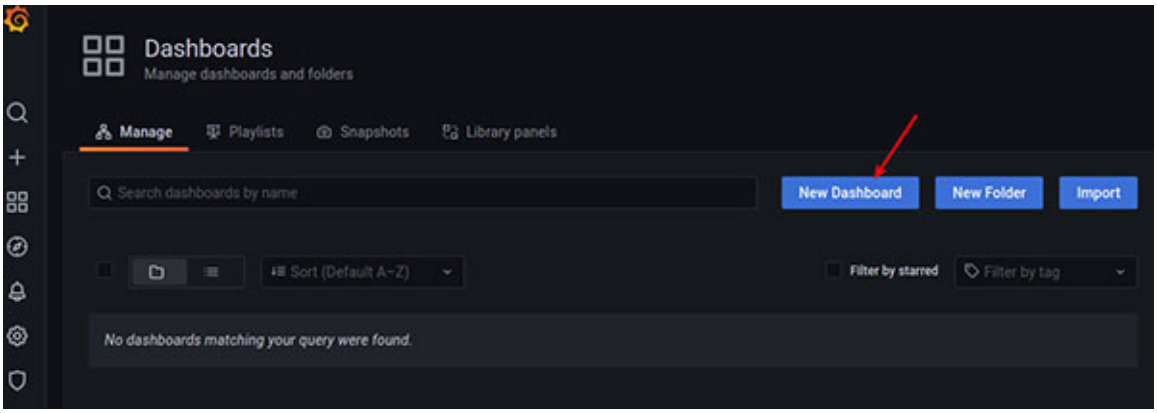


Figure 5.25: *Create a new dashboard for visualizing Sense Hat data*

9. Then, click on the empty panel rectangle as shown in the following figure:

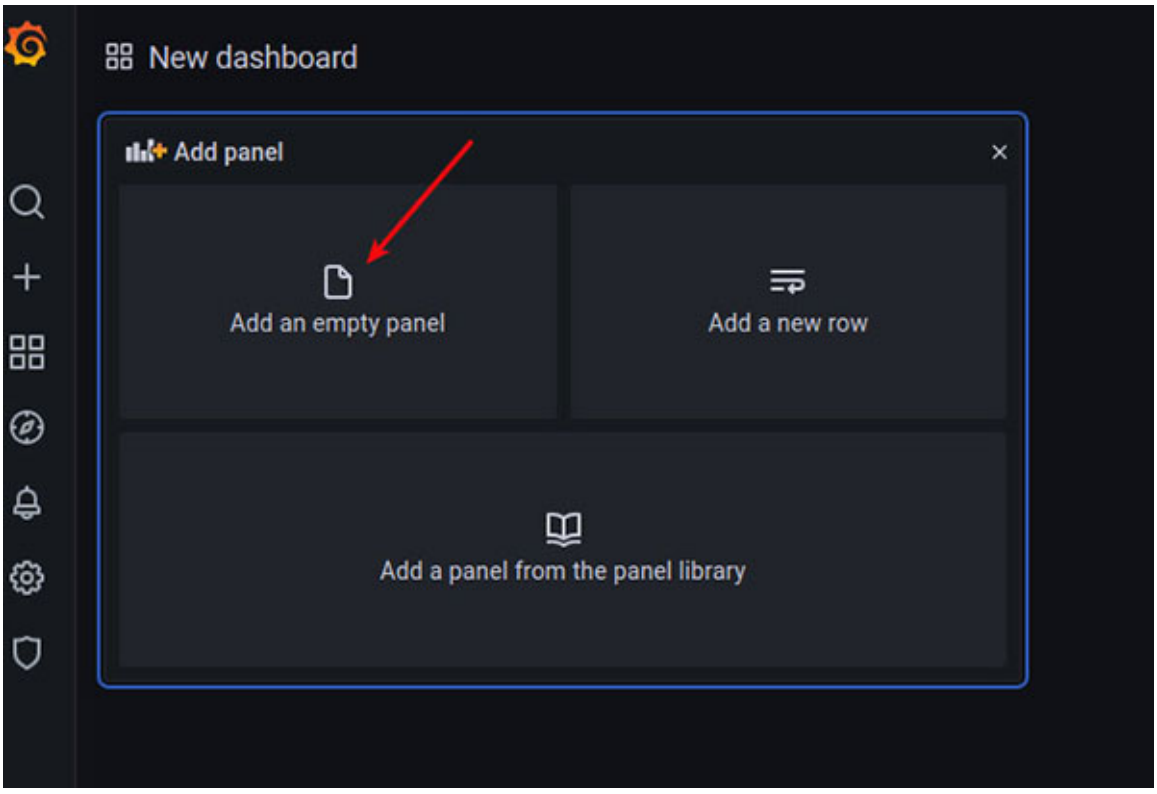


Figure 5.26: *Start from an empty panel*

10. Customize the query to use the measurement `sense_hat_avg_simulator` and select the `field (temperature)`. Then, give a title to the graph as shown in the following figure:

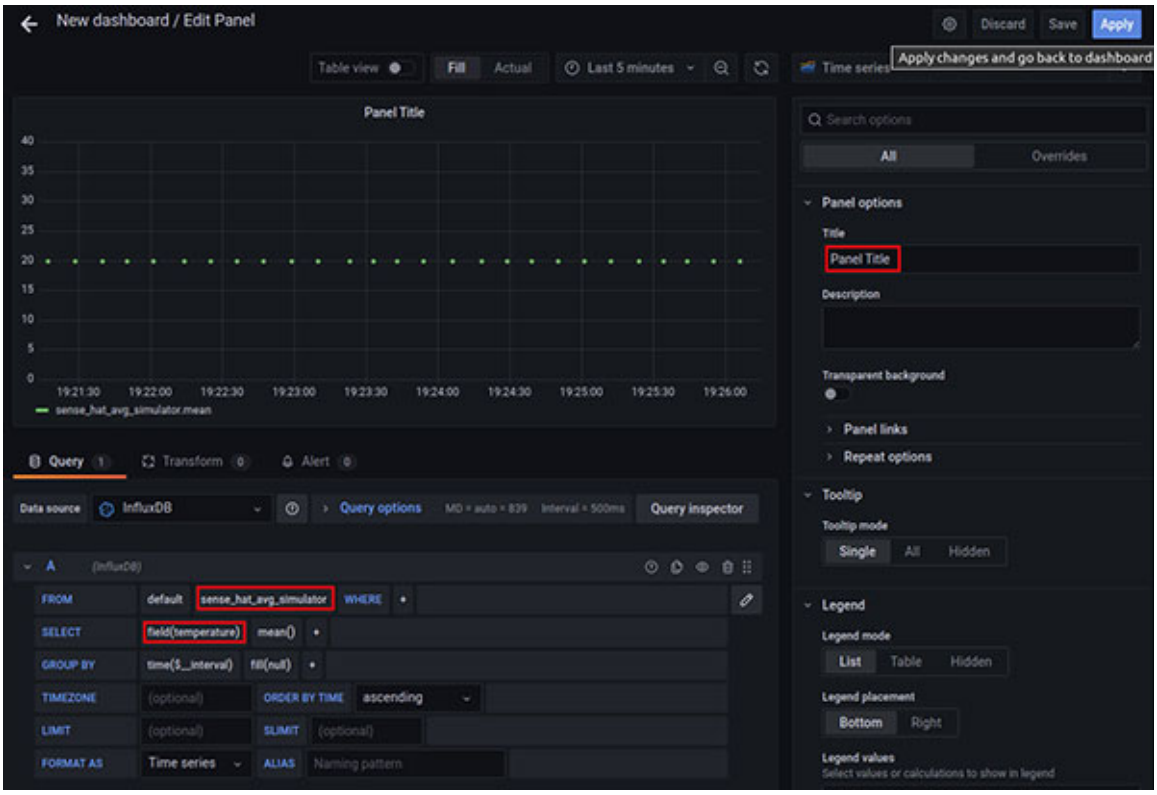


Figure 5.27: *Set up the graph of temperature coming from the Sense Hat simulator*

11. Save and see the result as a real-time graph (refer to *figure 5.28*). You will see the points and the X axis ticks moving to the left every time a new measurement comes (interval is 10 seconds as defined in the NodeRED flow as per throttle node in Store aggregated data flow).

Figure 5.28: *Real-time graph of temperatures from the Sense Hat simulator*

12. Access the Sense Hat simulator interface at **http://localhost:1880/sensehat-simulator** and manually raise the temperature to 25 degrees C, slowly as shown in the following figure:

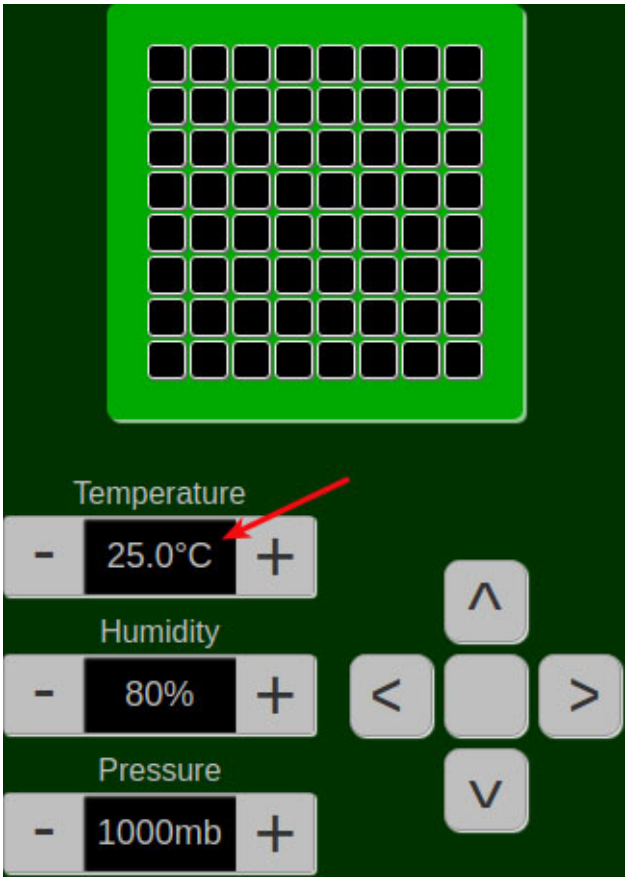


Figure 5.29: *Manually raise the temperature in Sense Hat simulator tab*

13. Every 2 seconds, in Node RED, a temperature value is acquired and averaged every 10 measurements. This average value is stored in InfluxDB every 10 seconds. The stored temperature data is read from Grafana. As a result, you will see the soft curve in the following figure:

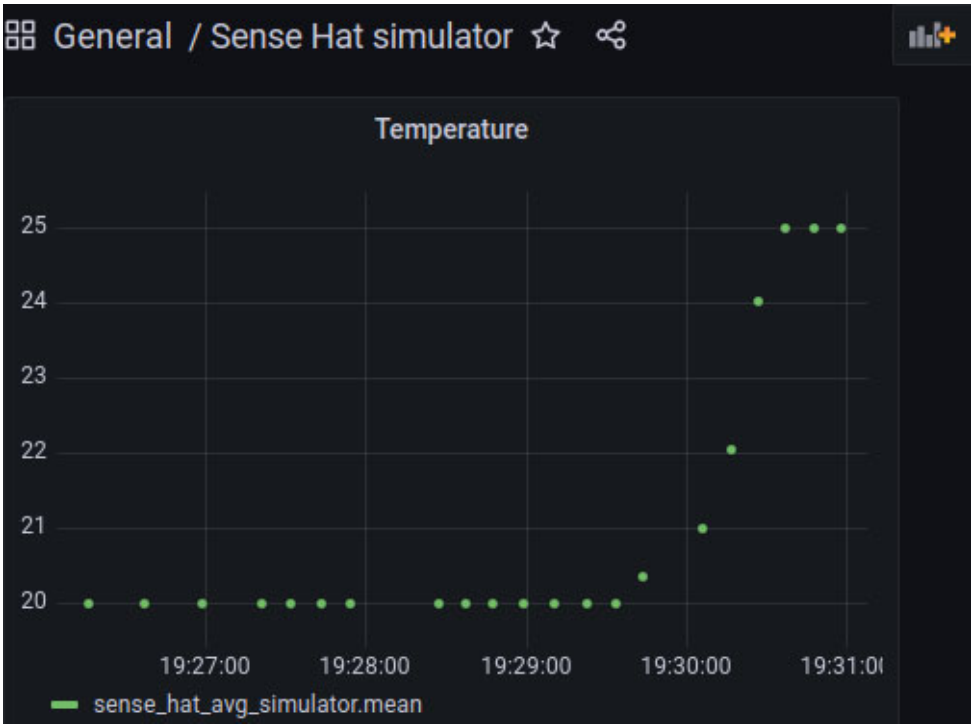


Figure 5.30: *Real time data point as temperature is rosen up*

At this point, we have covered all the software components to visualize temperature data coming from the Sense Hat simulator. In *Chapter 7, The IoT software package*, we will follow the same steps to represent data from the actual (hardware) Sense Hat.

Conclusion

In this chapter, we have covered the basics for persisting data in a safe storage, using a time series-oriented database like InfluxDB. This one offers clear technical advantages over general purpose NoSQL databases like MongoDB, because of the specific tools it provides for interacting with time series.

We first provided additional background on the software architecture of the application to show how the storage component fits in; then, we

detailed how to set up InfluxDB into your laptop; and finally, we proceeded to explain how to source the sensor data from the virtual Sense Hat in NodeRED to the database.

At this point, you are aware of the importance of persisting data in order to build a robust application.

All of this knowledge provides us with the required infrastructure to build a friendly interface for the end users. Finally, we have learnt how to produce a graph for real-time data stored in the database with the open-source software Grafana.

This chapter concludes the software development using a simulator of the IoT sensors' board, i.e., Sense Hat. The next two chapters apply what you have learned to the actual IoT project, where the physical hardware substitutes the simulator. *Chapter 6, "The IoT Hardware Package",* focuses on hardware integration and testing, while *Chapter 7, "The IoT Software Package"*, does the same for the software. The last has two parts, one running in the Raspberry Pi and the other in the remote server. The only new changes in the code are those regarding the Sense Hat data acquisition.

Points to remember

- NoSQL databases are different from classical SQL in that they do not impose a structure for the data a priori. This eases the interaction with huge amounts of data.
- InfluxDB is a NoSQL database optimized for dealing with time series.
- NodeRED provides contributed nodes to read and write to InfluxDB.

Multiple choice questions

1. **What is a NoSQL database?**

    a. one whose purpose is to collect time series of data

    b. one that does not have any schema

c. one whose data structure is not restricted as it is the case of relational databases
d. has the opposite meaning of SQL

2. **What is the specific purpose of the InfluxDB database?**
    a. manage Big Data applications
    b. focus in data streams where time is the main independent variable
    c. use for IoT applications
    d. use for the conjunction of Big Data and IoT applications
3. **What is the InfluxDB query language like?**
    a. It is very similar to that of MySQL.
    b. It is a variant of Java.
    c. It is a variant of JavaScript.
    d. It is similar to that of other NoSQL databases.
4. **What is a measurement in InfluxDB?**
    a. a time series of a given variable
    b. a data series of a given variable
    c. a time series of a sensor
    d. a collection of timestamps
5. **What timestamp is assigned to InfluxDB data points in a measurement?**
    a. the system time in all cases
    b. the system time provided by the device that is supplying the data
    c. the system time if none is specified in the write command
    d. the internal InfluxDB time

Answers

1. **c**

2. **b**
3. **a**
4. **a**
5. **c**

Questions

1. What is a NoSQL database?
2. What makes InfluxDB the database of choice for our IoT application?
3. Explain the several options we now have to query sensor data from NodeRED.

Key terms

- **NoSQL**: It is the acronym of Not Only SQL, identifying the type of databases in which a data schema is not required a priori.
- **Data storage**: It is the functionality of an application that persists its data in a permanent store.
- **Time series**: It is a list of pairs where each one is composed of a timestamp and the values of a set of variables. In the scope of InfluxDB, they are represented as measurements (see the next definition).
- **InfluxDB measurement**: It is the values of a set of variables that are recorded over time. You can think of measurements as a table where timestamps are the first column, and the rest are tags and fields.
- **InfluxDB tag**: They are indexed columns that allow performing data searches on the database. They are of string type.
- **InfluxDB field**: They are non-indexed columns that contain the values of the recorded variables. They are of numeric type.

Part - III

Hands on IoT Programming

CHAPTER 6

The IoT Hardware Package

Introduction

This chapter is the starting point of working with the actual hardware, not needing the Sense Hat simulator. It describes the *IoT Foundation Layer* of the project and covers the practical development to set up your own hardware.

Up to this point, we skipped the direct usage of the IoT hardware because we wanted to progressively build the application's software. For this reason, we built a virtual environment that removed the complexity of sensor integration with the Raspberry Pi. This is what happened until the end of *Chapter 5, Storing and Graphing Data Streams with InfluxDB and Grafana*. The IoT hardware—composed of the Raspberry Pi and the Sense Hat—is finally introduced in this chapter. We will explain how to make them work together. We will also write a small software for testing that the environmental sensors of the Sense Hat work as expected.

You will learn that testing the hardware integration is an essential step so that a complex application can run on top of it, and that the correct operation of the application is a matter of integrating software pieces that were developed separately. In summary, for the software-developer role in part two of the book, this part three will push us to develop our skills as software integrators.

As a result of following the contents in this chapter, you will set up the IoT hardware in order to get it ready for the software integration covered in the last two chapters of the book.

Structure

In this chapter, we will discuss the following topics:

- The application architecture
- The IoT hardware integration
- IoT hardware unit testing

Objectives

After studying this unit, you should connect the Sense Hat to the Raspberry Pi via the GPIO and make the Sense Hat communicate with the Raspberry Pi.

Technical requirements

We will use the actual hardware of the application, so we will divide this section into separate subsections to identify each category of the requirements.

Hardware

Apart from the laptop you have being using, you will need:

- a *Raspberry Pi* board. We recommend using the *version 4*, although you can use the cheaper Raspberry Pi 3. When there is a difference in the instructions depending on the Pi version, we will notify it in the explanations. Together with the board, you should get a micro-SD card where you will burn the operating system (use an SD card of class 10 at least, for obtaining acceptable writing speed). The list of materials for the Raspberry Pi assembly is as follows:
    - a Raspberry Pi 4 board whose official reference can be found in this URL: **https://www.raspberrypi.org/products/raspberry-pi-4-model-b/** (As mentioned above, version 3 is fine as well, and its URL is **https://www.raspberrypi.org/products/raspberry-pi-3-model-b/**)
    - a USB-C cable for power supply, rated at 3A minimum

    - a 5V power supply providing 3A at least (For example, this one referenced in the Raspberry Pi website at **https://www.raspberrypi.org/products/type-c-power-supply/.)**
- a *Sense Hat* board, to be plugged on top of the Raspberry Pi GPIO. The official reference can be found at **https://www.raspberrypi.org/products/sense-hat/.**

For the hardware, you should recap the section *Getting to know the Sense Hat* from *Chapter 4, Real time Data Processing with NodeRED*.

The software will run in the Raspberry Pi. You will not need to do any software operation for the Sense Hat since it comes with its firmware preloaded.

Software

The code for this chapter is in the GitHub repository that you can find at **https://github.com/Hands-on-IoT-Programming/chapter6**

Later, you will be instructed to clone the code at the right moment in order to learn how to deploy it to the Raspberry Pi.

Cloud services: Pusher and Balena

This subsection refers to the managed services—provided by third parties—on which the application will rely. This was the case for Pusher for the data streaming introduced in *Chapter 4, Real time Data Processing with NodeRED*. This service will not be necessary in this chapter, but in the next one.

We will now implement a new cloud service called *Balena*, conceived to manage the software in IoT devices like the Raspberry Pi. We will provide the introduction to Balena at the beginning of the section *IoT hardware integration*. Anyway, the curious reader can explore it now by himself visiting its web page at **https://www.balena.io/what-is-balena.**

As a didactic prerequisite, we will provide a detailed description of the software architecture of the whole application. It is essential to conceptually understand it, so that you make sense of the modular developments we carried out in the previous chapters and acquire insight into what is to be done in this last part of the book.

At this point it is worth that you recap the software architecture for IoT projects that we described in *Chapter 1, Introduction to IoT Applications and Their Software Architecture*.

Application architecture

Our IoT hardware includes a single board computer like Raspberry Pi, and a general purpose sensor board like the Sense Hat, that will allow us to skip the complexity of wiring and connecting individual sensors.

Since we built and tested the software in the previous chapters, we do not expect issues in the software itself. If any, they will arise as a result of including the actual hardware. Its integration occurs at two levels:

- **IoT hardware**, composed of the Raspberry Pi and the Sense Hat
- **Remote hardware**, for the purpose of the practical examples in this book it will be your laptop, but in general it will be a Linux server

This point lets us practically introduce the software layers formulated in *Chapter 1, Introduction to IoT applications and their Software Architecture* and recap the *figure 1.5* of that chapter.

Hence, this final part of the book deals with integrating the software pieces between themselves and with the actual hardware.

The IoT foundation layer

Taking into account the two levels of hardware integration formulated above, in the case of the *IoT hardware*, we have to

ensure that:

- *6.1 The Sense Hat connects properly to the Raspberry Pi via the GPIO*
- *6.2 The Sense Hat drivers are installed in the operating system of the Raspberry Pi*
- *6.3 The IoT device (Raspberry Pi) is properly managed from Balena cloud*
- *3.1 Sensor readings are retrieved from the Sense Hat*

These goals, technically known as *requirements*, are in the scope of this chapter.

Requirements are numbered with the following logic:

```
<CHAPTER_NUMBER>.<requirement_number>
```

This way, when we reference a requirement in the coming chapters, you will know what chapter it is derived from:

- **Requirements 6.1, 6.2 and 6.3 are derived from this same chapter.**
- **For 3.1 in the list above, the requirement is derived from the work we did in *Chapter 3, Data Acquisition and Real-Time Streaming*.**

On the other side, for the part of the *remote hardware*, we have to ensure that the code developed in *Chapter 3, Serving real-time data with Pusher*, runs properly in the Raspberry Pi. This includes:

- *3.2 Environmental data acquisition from the Sense Hat using NodeRED*
- *3.3 Streaming data to the Pusher app in the cloud*

Requirements 3.2 and 3.3 are in the scope of *Chapter 6, The IoT Software Package*.

The middleware layer

This layer is deployed on the *remote hardware*, and the requirements it has to satisfy are as follows:

- The code developed in *Chapter 4, Real-Time Data Processing with NodeRED* runs properly in the laptop (acting as a remote server). This includes:
    - 4.1 Receiving the environmental data from the Pusher app
    - 4.2 Obtaining their basic statistics in the laptop
- The code developed in *Chapter 5, Storing and Graphing Data Streams with InfluxDB and Grafana*, also runs in the laptop. This includes:
    - 5.1 Storing the environmental data in the InfluxDB database
    - 5.2 Grafana connecting to the InfluxDB database
    - 5.3 Showing the Grafana dashboard via the web browser

The requirements 4.1, 4.2, 5.1, 5.2, and 5.3 are in the scope of *Chapter 7, The IoT software package*.

The application layer

This layer is also deployed on the remote hardware and the user interface is served on every client device (laptop, smartphone, tablet, etc.). The requirements it has to satisfy are as follows:

- The code developed in *Chapter 5, Storing and Graphing Data Streams with InfluxDB and Grafana* runs properly in the laptop when accessing the Grafana dashboard from the web browser. Alternatively, you could use another PC or smartphone connected to the same network. The requirements are:
    - 5.2 Grafana connecting to the InfluxDB database
    - 5.3 Showing the Grafana dashboard via the web browser

These are in the scope of *Chapter 7, The IoT software package*.

In the following sections, we will cover the setup and testing of the IoT Foundation Layer.

IoT hardware integration

This section focuses on the Raspberry Pi and Sense Hat integration, as well as their connection to the Balena cloud service. The section after this one will cover the unit tests to check if the hardware is working properly.

This section covers requirements 6.1 and 6.2 as stated in the previous section:

- *6.1 The Sense Hat connects properly to the Raspberry Pi via the GPIO*
- *6.2 The Sense Hat drivers are installed in the operating system of the Raspberry Pi*

Before setting up Balena with the Raspberry Pi, make sure that the Sense Hat is plugged on top of the board attaching it to the GPIO.

The Balena platform

This is the second cloud service that is going to be part of the application. Recap that the other is Pusher, the messaging app that allows us to stream sensor data from the Raspberry Pi to the remote server.

Balena is a cloud-based infrastructure for developing, deploying, and managing fleets of IoT devices at scale. Hence, it is an enterprise-grade service that provides a complete IoT solution.

For our case of individual users, we should not worry about the price, because Balena offers all its functionality for free up to a maximum of 10 devices in a single account. This makes it easier for developers to get familiar with the platform and evaluate its potential adoption at enterprise level.

Another essential reason to choose Balena for the book's project is that its base code is open source. This point serves as the basis for introducing its architecture:

- On one side, there is the Balena server, that is provided as a managed service in the cloud (visit **https://www.balena.io**), or as the open-source version to install in your local infrastructure, i.e., **https://www.balena.io/open**. The cloud version provides an http user interface that facilitates the management of devices from a visual dashboard.
- On the other side, there is BalenaOS (visit **https://www.balena.io/os**), a Linux operating system optimized for embedded devices. Apart from having a small footprint, its essential feature is that it is focused on working with Docker containers, the most extended technology nowadays for packaging applications and distributing them under a continuous integration approach.

  Note: Learning to create and manage applications with Docker is a topic that requires a whole book for itself. Docker, also an open-source technology, is a tool oriented to developers, so it works at a relatively low level. What the Balena platform provides is an optimized Docker engine for IoT devices, as well as a wrapper so that users do not have to deal directly with Docker commands. This way, you will quickly learn to deploy applications without needing to master Docker before. In conclusion, all the complexity of managing these Linux containers is abstracted under Balena CLI (see the next bullet).

- The last component is Balena CLI (visit **https://www.balena.io/docs/learn/more/masterclasses/cli-masterclass**), the command line tool that allows deploying to the fleet an application that is developed and tested locally.

Given this brief overview of the Balena architecture, let's take a hands-on approach for deploying our IoT hardware on the cloud platform.

Account setup

The first step is to create a free Balena account at **https://dashboard.balena-cloud.com/signup**. This service provides the first 10 devices with all the functionality available on the platform for free. So, it is an ideal way to have a feeling of how a production environment will work.

Create an application

After logging in the platform, you will be redirected to the application's dashboard, that will look empty like this:

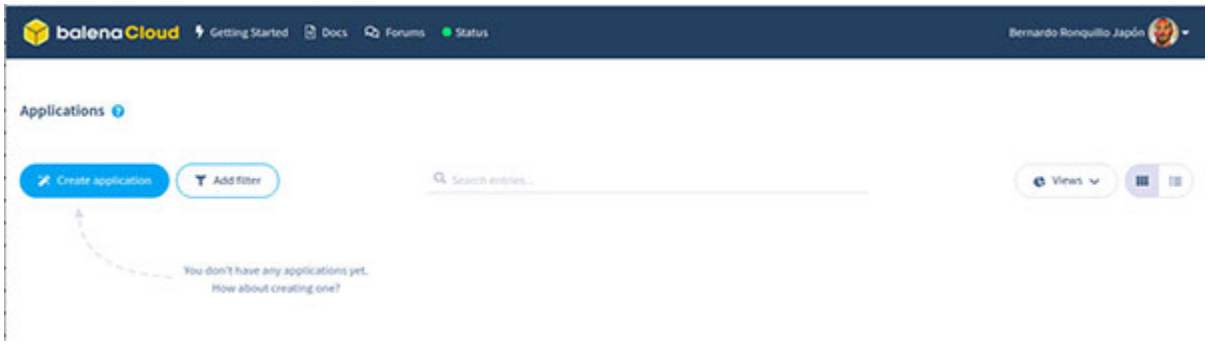


Figure 6.1: *Balena screen for application's dashboard*

Click the `Create new application` button, and fill in the form that will pop up, specifying the name for the app (i.e., Sense-Hat-Startup) and the default device type (Raspberry Pi 4 in the image below). Choosing Raspberry Pi 4 means that the BalenaOS operating system running in the board needs to be compatible with 64 bits CPU architecture, i.e., arm64 type. For the other field, the `Application type`, set it to the default 'Starter':

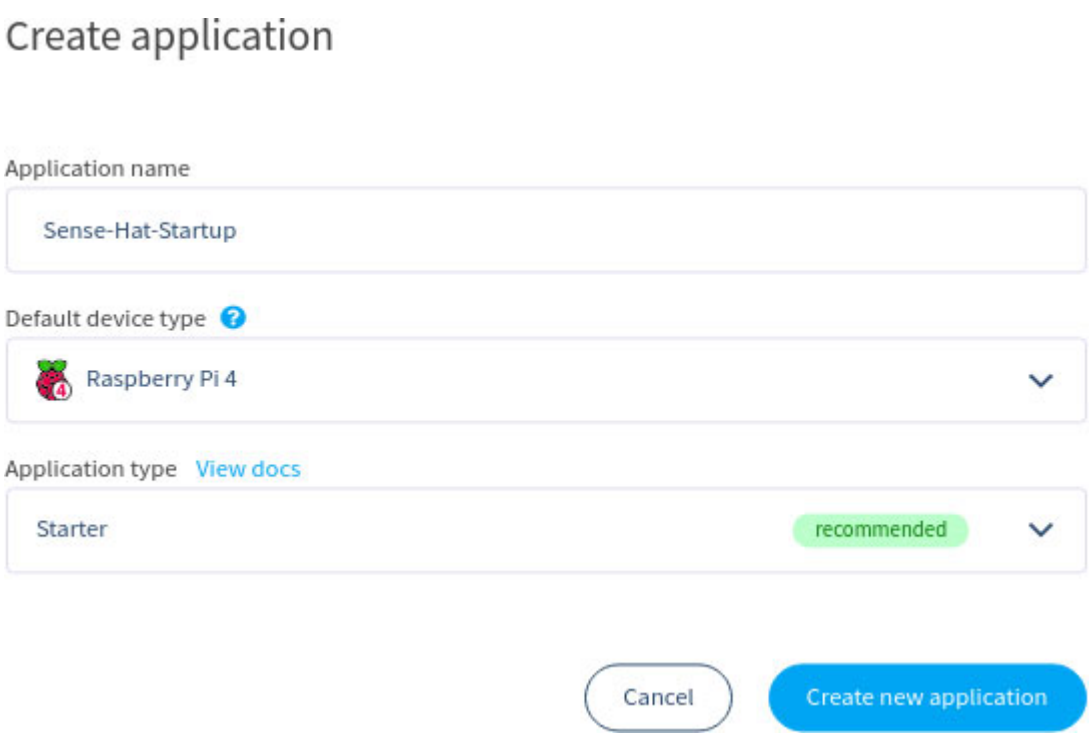


Figure 6.2: *Balena screen for setting up a new application with Raspberry Pi 4 (arm64 CPU architecture)*

If you are using a Raspberry Pi 3, the only difference is the selection of the device type, as shown in the following figure:

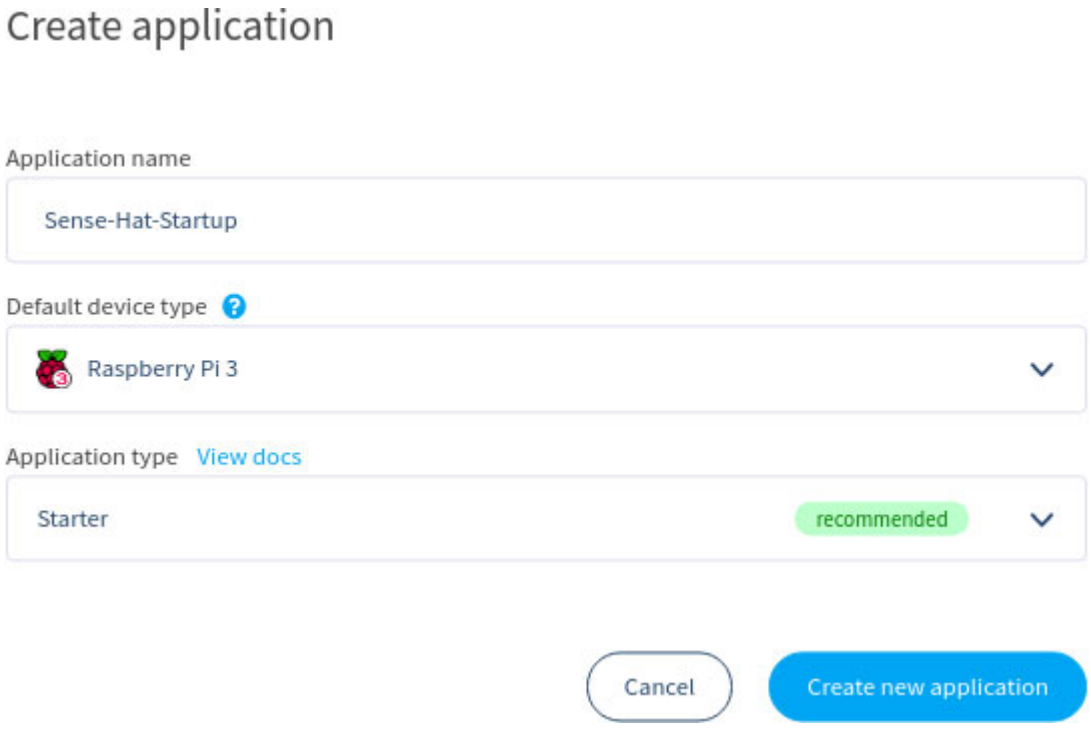


Figure 6.3: *Balena screen for a setting up a new application with Raspberry Pi 3 (armhf CPU architecture)*

Be aware that if you are choosing a Raspberry Pi 4 as the default device type, you are constraining just the CPU architecture to be arm64 (64 bits). When you add a second device, you will be presented with the list of devices that can run under an arm64 operating system. Then, for Raspberry Pi 3 the item in the dropdown list will be Raspberry Pi 3 (using 64bit OS), instead of only Raspberry Pi 3, in which case it runs armhf (32 bits) BalenaOS operating system (remember that both Raspberry Pi 3 or 4 can run any of 32 bits or 64 bits operating systems).

Tip: Summarizing, the only limitation you are assuming when selecting the default device type is the chipset architecture, armhf (32 bits) vs. arm64 (64 bits). This imposes a restriction on the base images to be used for the Docker containers. If you use a 32 bits BalenaOS, the containers shall be built on top of a 32 bits base operating system.

The code you are developing could be built on top of any of armhf or arm64 architectures. If you plan to sense several sensing locations using different device types—possibly mixing CPU architectures—it is recommended that you make a separation in different Balena applications, one for 32-bits architecture and another for the 64-bits devices.

If you are using Raspberry Pi 4, you should not face any incompatibility with the base image, because Balena OS for this device is able to run both 32-bits and 64-bits applications. See an example in this URL https://www.balena.io/blog/run-a-mixed-architecture-app-using-64-bit-balenaos-on-the-raspberry-pi-4

After creating the application, you are ready to attach the first device.

Adding your IoT device

Press the **`Add device`** button in the page of the created application Sense-Hat-Startup, as shown in the following figure:

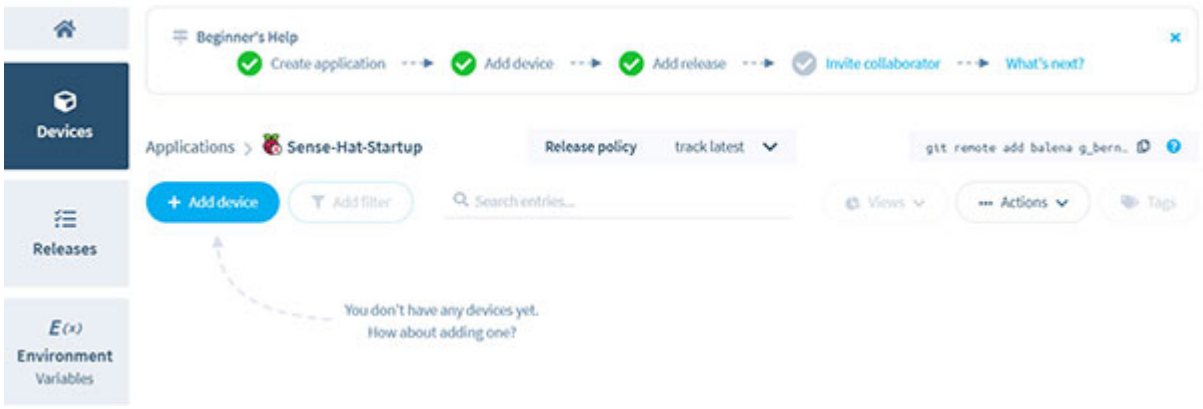


Figure 6.4: *Add a device to the Sense-Hat-Startup application*

You will be presented with a pop up window, like the one shown here:

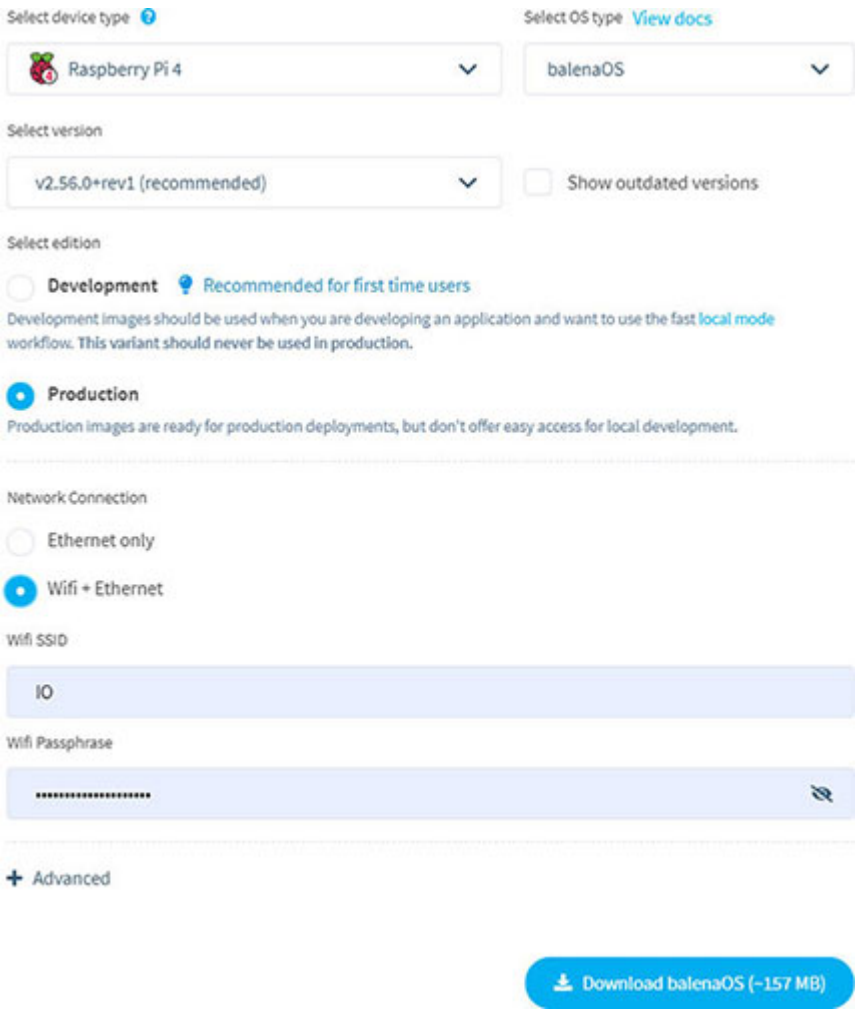


Figure 6.5: *Configuration window for the device to add to the Sense-Hat-Startup application*

Specify your local Wi-Fi credentials, and then download BalenaOS pushing the button on the bottom-right part of the window.

Notice that the operating system's footprint is really minimal: only 157 MB compressed, which expands to 913 MB when burned to the micro SD card. In comparison, the official operating system for Raspberry Pi is a 1.7GB download that expands to 4.8GB in the micro-SD card.

Provision of the device

Once the image of BalenaOS is downloaded, you can burn it to a micro-SD card using the open-source tool Etcher that you can

download from **https://www.balena.io/etcher**.

If using this tool, you do not have to decompress the image before burning it, Etcher does it automatically for you as part of the writing process.

After BalenaOS is burned, insert the micro-SD card into the slot in the Raspberry Pi 4. Make sure that the Sense Hat is attached to their GPIO, then connect the Raspberry Pi to the internet using a network cable. If you chose the Wi-Fi + Ethernet option (see *figure 6.5* above), you don't need the cable. In this case, once you power up the board, it will automatically connect to the Wi-Fi access point you configured. After one minute or so, the device should appear in the Balena application dashboard.

Tip: You can connect the Raspberry Pi to an HDMI screen to see the progress of the connection process. At the end of it, you should see the message "*Booted - Check your balena dashboard*" on the screen. If, instead, you see a black screen with 4 raspberries or rainbow colors, what happens is that the micro-SD card is corrupted or was not burned correctly.

Then, let's deploy some code to the device.

Installing Balena CLI

As mentioned in *The Balena platform* subsection above, Balena CLI is the tool for managing devices, that includes the deployment of the code developed locally.

In order to install it, choose the version corresponding to the operating system of your laptop from the releases page at **https://github.com/balena-io/balena-cli/releases**.

Now, let's describe the process for installing it in your laptop:

1. Download the standalone release for Linux from **https://github.com/balena-io/balena-cli/releases/download/v12.14.11/balena-cli-v12.14.11-linux-x64-standalone.zip**

2. Unpack the content, and place it in the root of your home folder. If you look at its structure, you should see something like this:

```
$ tree opt/balena-cli/ -L 1
opt/balena-cli/
├── balena
├──  @balena.io
├──  drivelist
├──  ext2fs
├──  lzma-native
├──  mountutils
├──  net-keepalive
├──  xdg-open
├── xdg-open-402
└── xxhash
```

   The file marked in bold letters, i.e., **`balena`**, is executable to run the CLI.

3. Add this folder to the **`file/etc/environment`** that will allow the **`balena`** command to be accessed system wide:

```
$ sudo nano /etc/environment
PATH="/home/ubuntu/balena-
cli:/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbi
n:/bin"
```

4. Then, source the file so that the **`PATH`** variable is updated now (so that you do not need to logout or reboot the system now):

```
$ sudo source /etc/environment
```

Alternatively, you could set the Balena PATH only for your user by adding it to the **`.profile`** file that is in your home directory:

```
$ sudo nano ~/.profile
PATH="/home/ubuntu/balena-
cli:$HOME/bin:$HOME/.local/bin:$PATH"
```

Then, source the file as in the equivalent step 4 above:

```
$ sudo source ~/.profile
```

Tip: You can explore all of the available commands of Balena CLI at the URL https://www.balena.io/docs/reference/balena-cli/. There are commands regarding applications, devices, authentication, environment variables, logs, and network. It is worth having a look at it to acquire an overview of all the utilities it provides.

Type in `balena help` in a bash terminal to check that the command is recognized. If working properly, go to the next subsection, where you will deploy the local code to the Raspberry Pi.

Push the application code

The first Balena CLI command you we will run is `balena login`, because prior to any interaction with the Raspberry Pi you need to be authenticated locally to your Balena account:

```
$ balena login
```

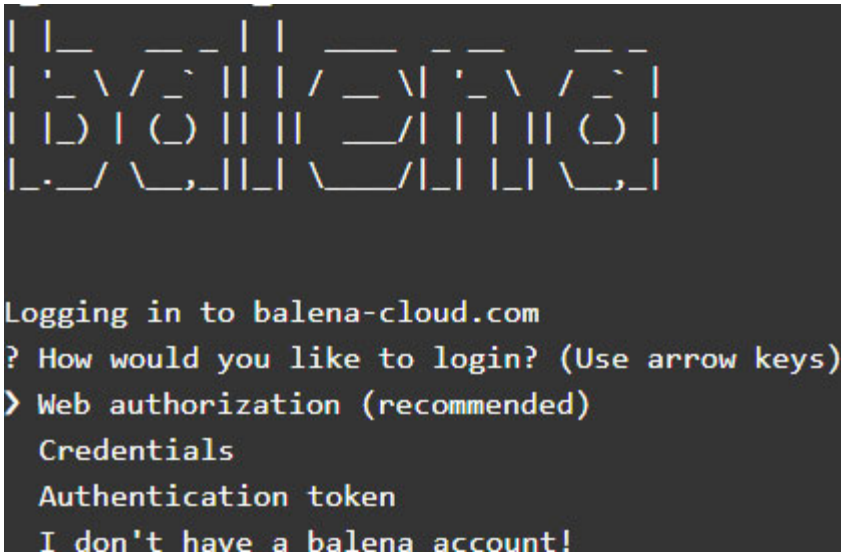


Figure 6.6: *Authentication options for Balene CLI tool*

If using the recommended Web authorization option, you will be redirected to a browser window where you will be logged into the

platform and explicitly asked for authorization.

Tip: You can alternatively use the Credentials option that keeps the authorization process in the terminal. You will be prompted to introduce your username and password. If you created your Balen a account using a social login (Google or GitHub), you will first need to create a password. To do so, visit your Balena dashboard, select Preferences from the dropdown menu that appears when you click on the top-right icon of your user. Then, in the Account details tab, set the password.

Clone the repository of the chapter as usual:

```
$ cd ~/book_hands-on-iot
$ git clone https://github.com/Hands-on-IoT-
Programming/chapter8.git
```

Then, change to the folder of the repository and deploy the code to the Balena application:

```
$ cd ~/book_hands-on-iot/chapter8
$ balena push Sense-Hat-Startup
```

Note: There is an alternative method that does not require the Balena CLI, just using git commands is enough. It consists of adding the remote repository of the Balena application. You can find the command at the top-right part of the application page. Copy it and paste into your terminal:$ git remote add balena gh_therobotacademy@git.balena-cloud.com:gh_therobotacademy/sense-hat-startup.git

Yours should be different because it includes the route to the application in your Balena account. Afterwards, push the local code to Balena servers:

```
$ git push balena master
```

This command will build the application and push the container to the connected devices, i.e. The Raspberry Pi in our case.

Once more, Balena CLI eases your work by wrapping the complexity of git commands in one only command, i.e. balena push <APP_NAME>.

After the first part of the deployment process, the log messages in the terminal finishes with a successful Docker image upload. Then, the second part can be followed from the dashboard of the application (see *figure 6.7*). In this part of the process, the image is downloaded to every device connected to the application:

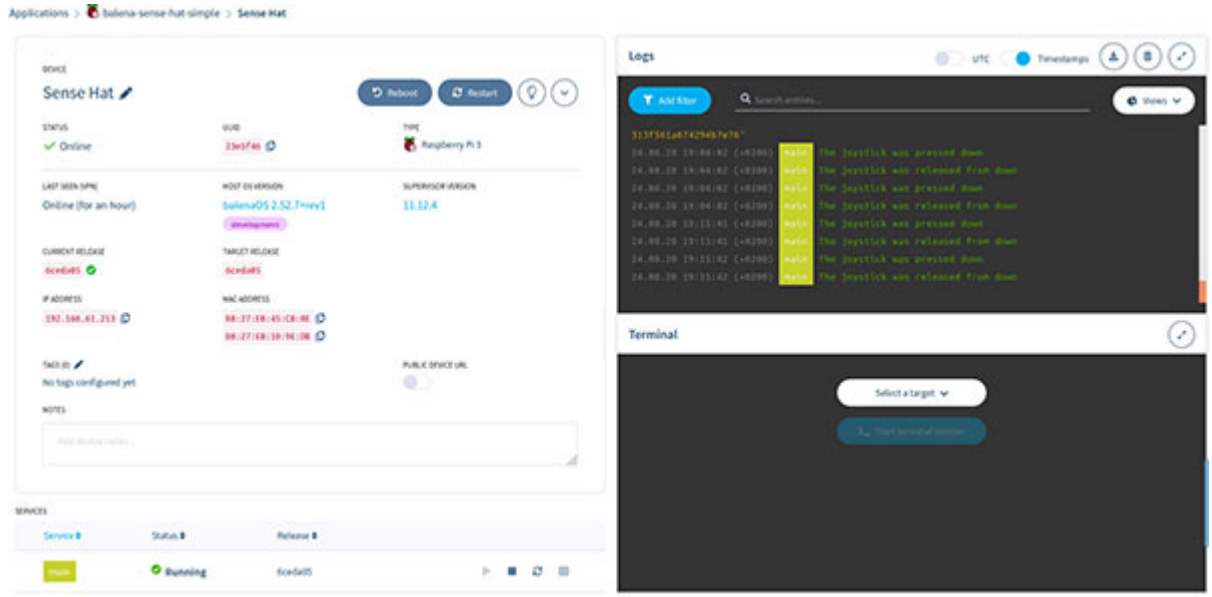


Figure 6.7: *Dashboard of the Balena app for testing the Sense Hat*

After downloading the image, the service that runs inside a Docker container is launched. You can follow the progress through the log of the application in the Logs window (upper-right one in the image above).

The code that runs in the service contains a testing application that we will explain in the next section.

IoT hardware unit testing

Now, we want to check that our assembled hardware works properly after the integration we carried out in the previous section.

Software testing can have different scopes, i.e., we can test a piece of code to validate it as follows:

- As an isolated software block called **code unit**. In this case, we say it is a unit test.
- Or, as a part of the software of the application. In this case, we call it a **functional test**, meaning that we will check whether the piece of code performs the functionality of the application it is expected to do.

Functional testing is related to the *functional requirements* of the application in the sense that such tests will have to show that, if they are successful, the software executed complies with that requirement.

Remember that for our project, the requirements have been defined in the section *Application architecture* above. Of all of them, there are four related to the IoT Foundation Layer. The first two were covered in the previous section, IoT hardware integration, while the rest will be verified in the next paragraphs.

Note: This section covers requirement 6.3 and 3.1 as stated in the section Application architecture above:

- 6.3 The IoT device (Raspberry Pi) is properly managed from the Balena cloud
- 3.1 Sensor readings are retrieved from the Sense Hat

So, what kind of tests do we have to run now: unit or functional? The answer is unit tests, because the software we have deployed is conceived to check the hardware as a unit, and that software need not be the one that finally runs in the application. Let's explain this point with the concrete use case: for the current scope of the IoT

Foundation Layer, we have to retrieve the temperature and humidity readings from the Sense Hat:

- In this chapter, we will use a simple script written in Python to check that requirement. Hence, we will be verifying the assembled hardware.
- But for the next chapter, we will implement the readings of the Sense Hat's sensors with a different piece of code, i.e., a NodeRED flow.

The Python code is not part of the application, while the NodeRED flow is:

- The Python script serves the purpose of checking that the sensors provide environment readings and they are acquired by the Raspberry Pi. This is a unit test of the hardware.
- The NodeRED flow serves the same purpose, being also part of the software of the application. Hence, it is a functional test of the software.

Having a clear understanding of the unit test we are going to carry out, let's explore what the Python code has to do.

Unit test definition

The workflow shown in the following figure explains what the test execution has to do:

1. Show colored stripes like a rainbow to check that the *LED matrix* of the Sense Hat works.
2. Select what sensor reading to show in the LED matrix using the joystick of the Sense Hat. A menu with four items will be depicted in the LED matrix: *temperature*, *humidity*, *pressure* and quit the script. You will have to push the joystick when the desired item is highlighted.
3. Move a dot on the LED matrix by moving the joystick.

4. Log environmental conditions (temperature, humidity, pressure) to the computer (Raspberry Pi) log.

In this list, the items in bold letters refer to the part of the hardware that is being verified in each test.

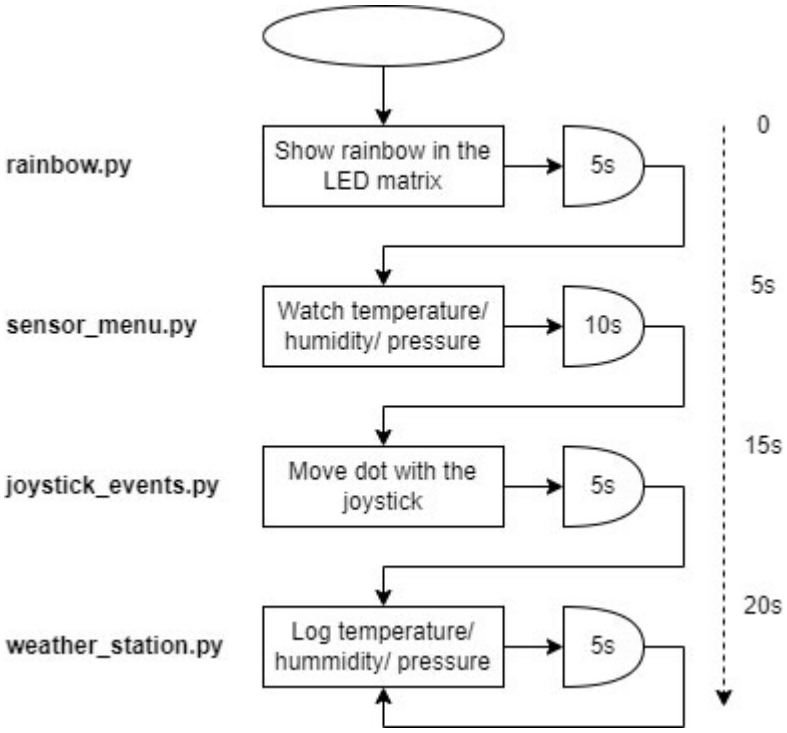


Figure 6.8: *Workflow for the IoT hardware unit testing*

To make sure that the tests run one after another, we introduce four delays with the execution of each one (5 seconds, 10s, 5s, and 5s, respectively). What appears to the right of each delay is the name of the Python script that performs each test.

Although, explaining the scripts in detail is out of the scope of this chapter (they are simple accessory blocks of code), what it is of our interest is to understand the structure of the files in the repository of this chapter. The following figure shows its tree structure:

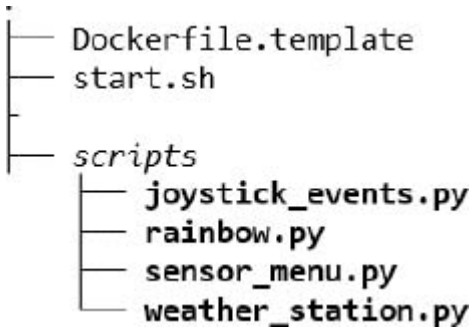


Figure 6.9: *Code structure for the IoT hardware unit testing*

Let's go one by one to understand what each file does:

- `Dockerfile.template` contains the definition of the execution environment (Python), and provides Balena the instructions to build the Docker container.
- The `scripts` folder contains the four code blocks identified in the *figure 6.9* above.
- The file `start.sh` is the bash script that is executed when the container is deployed to the Raspberry Pi. It runs all four Python scripts in a row.

To understand how the flow of tests happens, let's look at the content of `start.sh`:

```
echo "Watch the rainbow for 5 seconds to check I'm alive…"
./scripts/rainbow.py &
sleep 0.5
echo "You can always read any of temperature, pressure or
humidity with sensor_menu.py, using the Sense Hat joystick"
./scripts/sensor_menu.py &
sleep 0.5
echo "Move the joystick to test I'm still alive…"
./scripts/joystick_events.py &
sleep 0.5
echo "In parallel watch the environment conditions in the
container's log…"
./scripts/weather_station.py
```

You can guess that it simply executes the four Python scripts one after another. The '&' symbol after each line means that the Python program is sent to the background allowing for the delays needed to run one test after another in order. If we did not use the '&' symbol, every Python program would immediately overlap over the previous and we could not see how every test is completed.

There is an additional delay of 0.5 second after each line, so that the message logged when starting each test is shown in the right order. The execution delays of the tests (5 seconds, 10s, 5s, and 5s) are specified inside every Python script.

Unit test execution

Now, let's execute the application and have a look at the Log window in the Balena dashboard in the Logs window (located in the upper-right part in *figure 6.7*):

```
Installing service 'main
sha256:8fed59cb3c371a0ac4baacca9d04569d19f62942b48cfc338d08319
7e26df9b2'
Installed service 'main
sha256:8fed59cb3c371a0ac4baacca9d04569d19f62942b48cfc338d08319
7e26df9b2'
Starting service 'main
sha256:8fed59cb3c371a0ac4baacca9d04569d19f62942b48cfc338d08319
7e26df9b2'
Started service 'main
sha256:8fed59cb3c371a0ac4baacca9d04569d19f62942b48cfc338d08319
7e26df9b2'
03.10.20 10:47:03 (+0200) main
* Watch the rainbow for 5 seconds to check I'm alive…
* You can always read any of temperature, pressure or humidity
with sensor_menu.py, using the Sense Hat joystick
* Move the joystick to test I'm still alive…
```

```
* In parallel, watch the environment conditions in the
container's log…
- FIRST TEST: Show a dynamic rainbow on the LED matrix for 5
seconds
Exit Time of Rainbow 2020-10-03 08:47:09.864873 .. Now
clearing LED matrix
- End of FIRST TEST
-- SECOND TEST: Select with joystick for
TEMPERATURE/HUMIDITY/PRESSURE/Quit on the LED matrix
-- End of SECOND TEST (only if 'Q' item selected on time)
--- THIRD TEST: Move a dot on the LED matrix using the
joystick
--- This is an event oriented program, so it's always
listening to the joystick
---- FOURTH TEST: Show the TEMPERATURE/HUMIDITY/PRESSURE every
5 seconds
---- This is an infinite loop that runs concurrently with
THIRD TEST (move dot with the joystick)
03.10.20 10:47:26 (+0200) main {'temperature':
38.62160873413086, 'pressure': 1003.335205078125, 'humidity':
30.019060134887695}
03.10.20 10:47:31 (+0200) main {'temperature':
38.751617431640625, 'pressure': 1003.279296875, 'humidity':
29.908781051635742}
```

Reading the log, it is pretty clear when and how each test is executed. Verification of the Sense Hat requires you to physically interact with it moving/ pushing the joystick, and observe the LED matrix.

The first four lines of the log come from the Balena platform, and inform us about the deployment and start of the application in the Raspberry Pi.

Tip: The log messages have been carefully designed so that the user can understand, by just reading the Logs window, what the

software is doing in real time. Hence, if there is some error and the test is not passed, the messages should give enough information in order to know where the error comes from, and quickly fix it. The lesson to learn is how important it is to create a good log of your application to ease the debugging of the code.

If you want to restart the application, there is a button in the Balena dashboard to do so:

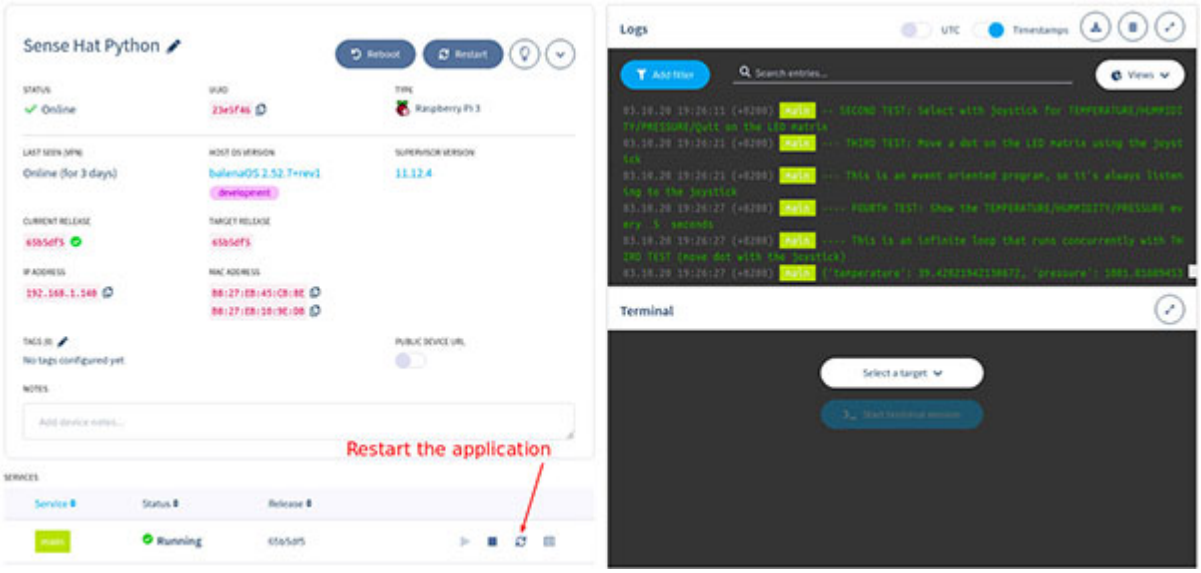


Figure 6.10: *How to restart the application from the Balena dashboard*

Although you should check it with the actual hardware, the following images show the behavior of the LED matrix for the sequence of Python scripts.

First, the `rainbow.py` script shows colored stripes on the LED matrix for 5 seconds:

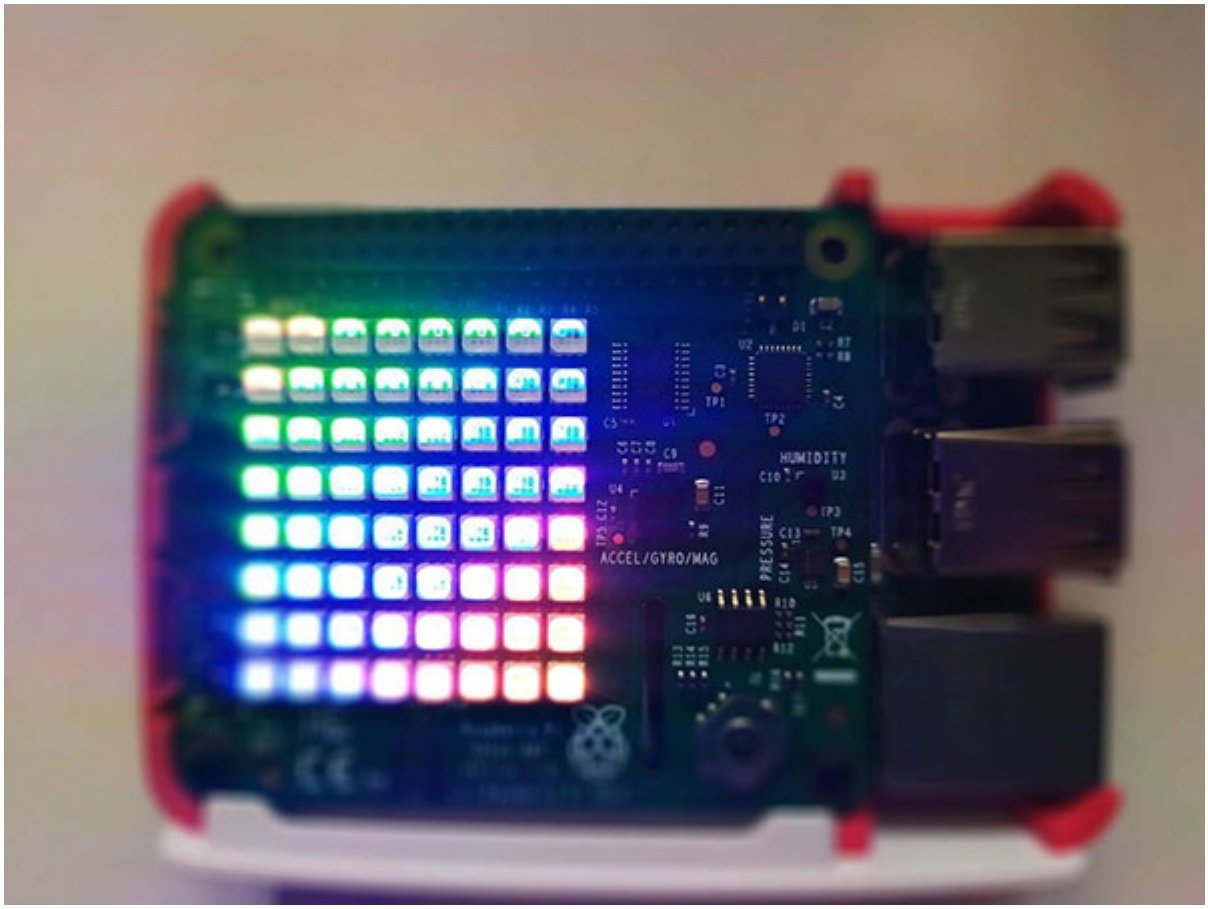


Figure 6.11: *The script rainbow.py produces a rainbow for 5 seconds*

Afterward, the `sensor_menu.py` script depicts on the LED matrix a menu for selecting temperature, humidity, pressure, or exiting the script as shown in the following figure. If the user does not select the Quit item within 10 seconds, the scripts exits automatically:

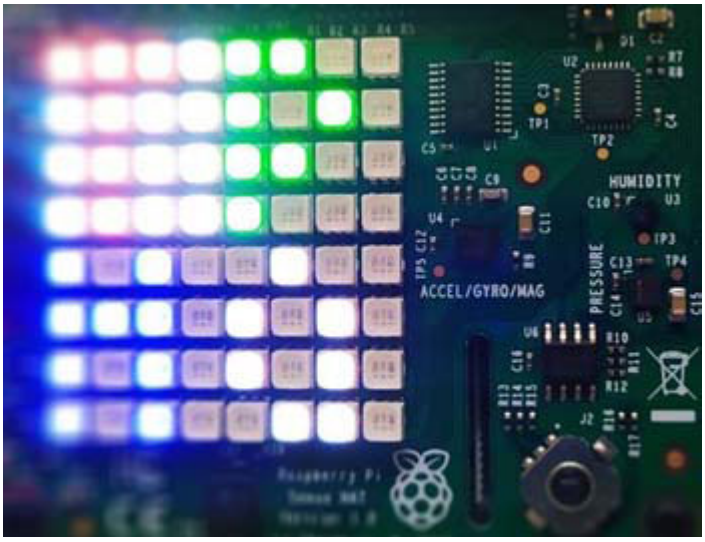


Figure 6.12: *The script sensor_menu.py shows a menu with 4 items that can be selected with the joystick*

Once exited from the `sensor_menu.py` script, the third test `joystick_events.py` comes into action for moving a white dot with the joystick.

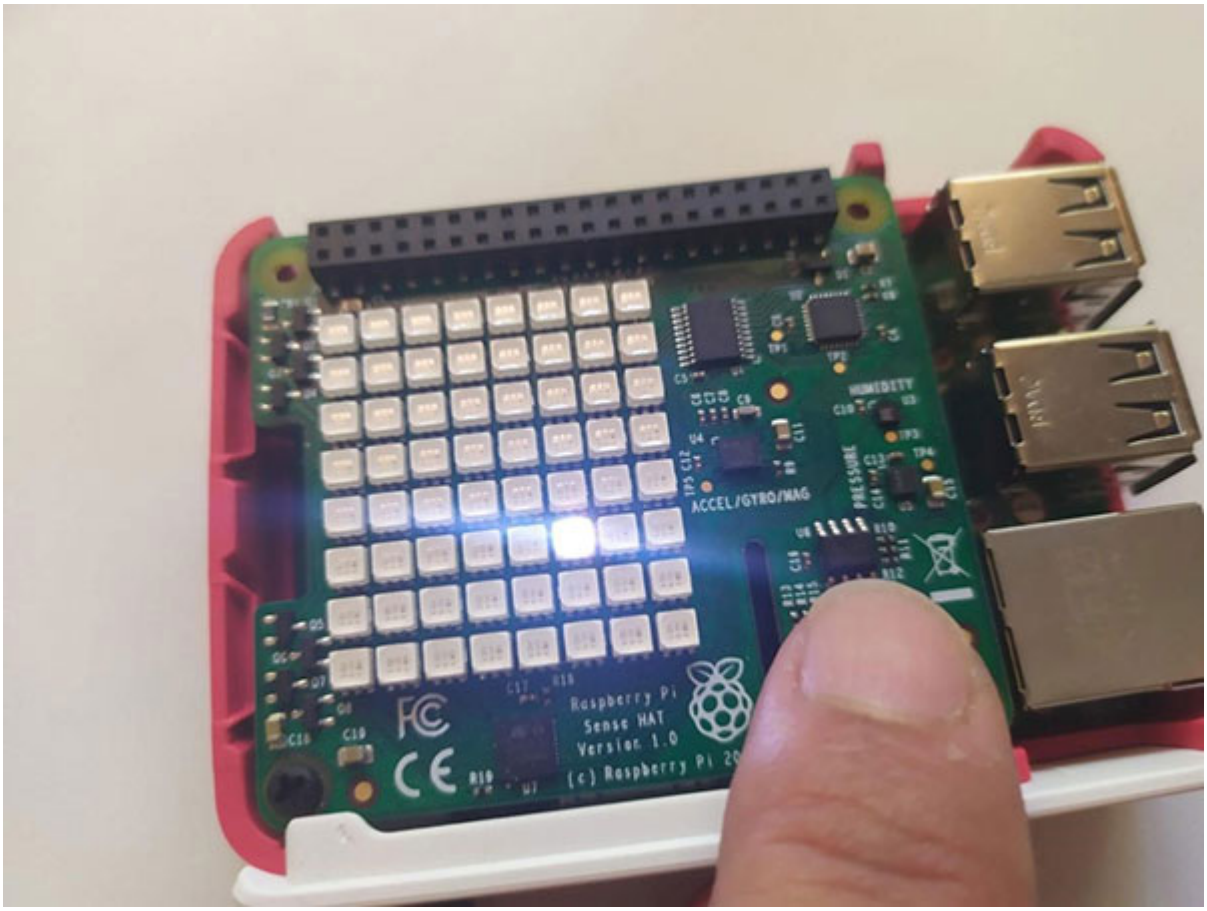


Figure 6.13: *Moving the joystick to displace the white dot when the joystick_events.py script is running*

Finally, after 5 seconds of starting the **`joystick_events.py`** script, the environmental conditions start getting reported in the Log window every 5 seconds due to the execution of the **`weather_station.py`** script. A sample log of messages covering 50 seconds (= 10 reading messages x 5 seconds/reading) is listed as follows:

```
{'pressure': 1003.1748046875, 'temperature':
45.19622039794922, 'humidity': 26.578964233398438}
{'pressure': 1003.177490234375, 'temperature':
45.21479034423828, 'humidity': 26.44417953491211}
{'pressure': 1003.17041015625, 'temperature':
45.02906799316406, 'humidity': 26.50544548034668}
{'pressure': 1003.176025390625, 'temperature':
45.177650451660156, 'humidity': 26.848535537719727}
```

```
{'pressure': 1003.184814453125, 'temperature':
45.19622039794922, 'humidity': 26.698434829711914}
{'pressure': 1003.16455078125, 'temperature':
45.1405029296875, 'humidity': 26.879169464111328}
{'pressure': 1003.187744140625, 'temperature':
45.19622039794922, 'humidity': 26.943498611450195}
{'pressure': 1003.16796875, 'temperature': 45.15907669067383,
'humidity': 26.9710693359375}
{'pressure': 1003.211181640625, 'temperature':
45.38194274902344, 'humidity': 27.099727630615234}
{'pressure': 1003.159423828125, 'temperature':
45.307655334472656, 'humidity': 26.81484031677246}
```

Hence, the timeline of the testing application is as shown in the next image, where we can see together with the workflow, the Python scripts and the timeline in the right-most part:

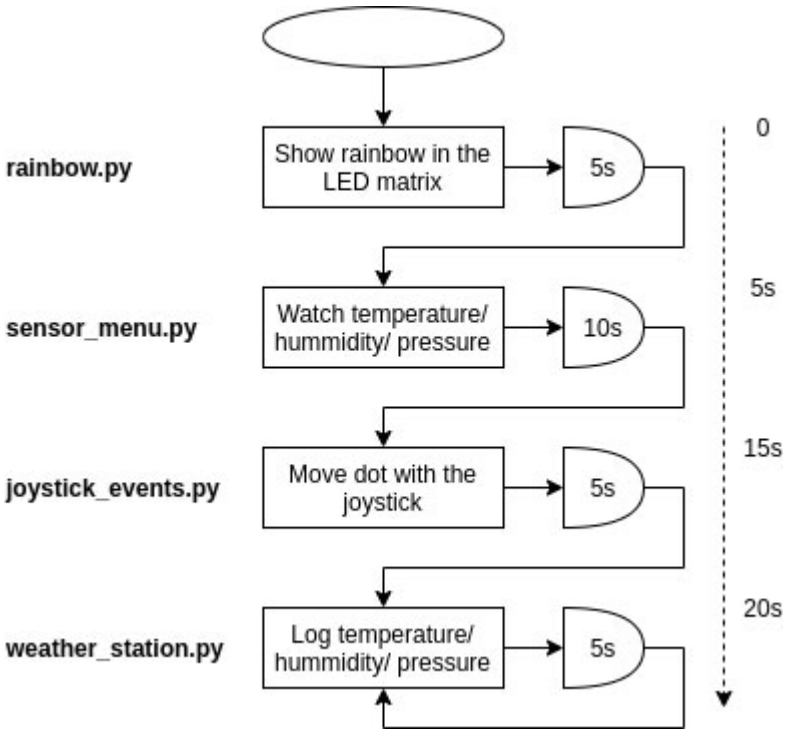


Figure 6.14: *Timeline of the testing application is shown in the right-most part*

At this point, if everything works as it should, the IoT hardware is ready

Conclusion

In Part 2 of the book, *Getting familiar with the Software Toolkit,* we covered the whole software development, and we tested with a simulator of the Sense Hat.

In this chapter, we have introduced the actual IoT hardware, i.e., Raspberry Pi and Sense Hat. By registering the board with the Balena cloud, we get to provide an infrastructure that allows us to manage IoT devices remotely: deploy new versions of the code, restart the application, reboot the device, solve issues remotely, and so on.

We have set up the hardware in such a way that the software tested with the Sense Hat simulator in the laptop can run on the Raspberry Pi as well. The validation of this assumption has consisted of carrying out unit tests to make sure that the Sense Hat measurements are acquired by Raspberry Pi.

This way, you have been trained in software engineering skills to set requirements for the application and carry out tests to check that the system is compliant with them.

This practice has been part of the more general approach of system engineering applied to software development that we have introduced in the chapter. Thanks to this background, you have learned how to write the requirements for the application, as well as carry out tests to check for compliance.

In the next chapter, you will deploy the code for two of the three layers, i.e., IoT Foundation and Middleware layers. This corresponds to the code developed from *chapter 3* through *5*. For each layer, you will validate the compliance carrying out their respective functional tests.

Points to remember

- To remotely manage IoT devices is an essential feature of applications where the code is distributed to different physical

locations. Platforms, such as Balena, serve this purpose both for individual developers as for industrial projects.

- Unit testing allows checking for proper working of isolated hardware/software modules.
- Functional testing allows checking for compliance of the application with respect to end-user requirements. It is a prerequisite that unit testing of all the modules involved are successfully passed.

Multiple choice questions

1. **What is the main advantage of running a part of an application in the cloud?**
    a. that it is a service managed by the provider, avoiding the need to apply upgrades and maintenance on the side of the application creator (you)
    b. that the server is in the cloud
    c. not clear if it's really more advantageous than hosting an equivalent open-source option on your own server
    d. that if your local server crashes, the service in the cloud would not be affected
2. **What is the difference between a unit test and a functional test?**
    a. Unit test applies to a single block of code, while functional verifies that a single function of the code works properly.
    b. Unit and functional are equivalent ways to designate software testing.
    c. Unit test applies to an isolated block of code, while functional verifies if an application level feature works as requested.
    d. They are subsequent steps in the software testing of the whole application.

3. **Could the Sense Hat be connected directly to the laptop for reading the environmental conditions?**
    a. No, because it needs to interface via the GPIO that is not an interface provided by laptops.
    b. Yes, you can attach it using a USB cable plugged to one of laptop's ports.
    c. No, because they have incompatible CPU architectures.
    d. It depends on the technical features of the laptop.
4. **What is the purpose of the log of an application?**
    a. to supply warning and error messages to the end user of the application
    b. to provide continuous data regarding the workflow of the application, for debugging and troubleshooting purposes
    c. to translate the code into a human-readable text
    d. to supply messages to the operating system reporting the health of the application
5. **What part of the code of the application is built in this chapter?**
    a. the part of the NodeRED flow that acquires the temperature and humidity from the Sense Hat
    b. the core functions that will be inserted in the NodeRED flow
    c. the user interface, i.e., the Balena dashboard of the IoT device
    d. none, the accompanying code is merely for checking that the IoT hardware works properly

Answers

1. **a**
2. **c**
3. **a**

4. **b**
5. **d**

Questions

1. What are the two cloud services that the application will rely on, and what is the purpose of each one?
2. What are the three components of the Balena cloud, and what is the purpose of each one?
3. Explain the relationship between application requirements and software testing.

Key terms

- **IoT hardware**: It carries arithmetic and logical operations.
- **Remote hardware**: It controls the operation to be performed.
- **Software requirement**: It is a statement of the characteristics that the code of an application, or part of that code, has to comply with
- **Unit requirement**: It is a statement of what a module of the application has to perform as an isolated block of code.
- **Functional requirement**: It is a statement of what the application has to perform at system level, i.e., integrated set of modules that compose it.
- **Unit test**: It is a test defined in order to verify the compliance of a unit requirement.
- **Functional test**: It is a test defined in order to verify the compliance of a functional requirement.

CHAPTER 7

The IoT Software Package

Introduction

In the previous *Chapter 6, The IoT Hardware Package*, we set up the hardware to deploy the software developed in part 2 of the book *Getting familiar with the software toolkit*. Along with those chapters, we developed the code of the application in a modular way. In the present chapter and the next one, we will take these code blocks and plug them together to build the final application.

This chapter describes and practically develops both the IoT Foundation and the middleware layers. Following these contents and carrying out the practical part, you will acquire insight into how the Raspberry Pi software is integrated with the different modules in the remote server: NodeRED for the processing of data and InfluxDB for storing time series of temperature data.

This is the core chapter of the book because this is where you will integrate most of the knowledge exposed so far, providing you with a standard way of working for doing so. Let us emphasize that software integration is as crucial as developing code because an application is functional only after all its components are integrated together.

Structure

In this chapter, we will discuss the following topics:

- IoT Foundation Layer integration
    - Deployment of the IoT layer in Balena application
    - Managing environment variables
    - Data acquisition with NodeRED
        - Splitting the flows between IoT device and remote server

    - Serving realtime data with Pusher
- Functional tests of the IoT foundation layer
- Middleware layer integration
    - Retrieving data from Pusher
    - Data processing with NodeRED
    - Storing historical data with InfluxDB
- Functional tests of the middleware layer

Objectives

After studying this unit, you should perform an end-to-end integration of two heterogeneous systems: an IoT environment with a remote server. Also, you should practically understand how the concept of a well-defined interface allows any modular software to talk with other modules via an **Application Programming Interface** (**API**).

Technical requirements

Hardware requirements are the same as for *Chapter 6, The IoT Hardware Package*:

- a Raspberry Pi 3 or 4 board connected to a 5V power supply
- a Sense Hat board is plugged on top of the Raspberry Pi GPIO

It's the software that is specific for this chapter.

Software

There are two repositories hosted in GitHub:

- One for the code to run in the Raspberry Pi, whose URL is:

  https://github.com/Hands-on-IoT-Programming/chapter7-balena

  This module corresponds to the IoT Foundation Layer of the application.
- And another to run in the remote server, whose URL is:

https://github.com/Hands-on-IoT-Programming/chapter7

This module corresponds to the IoT Middleware Layer of the application.

Later you will be instructed to clone each repository code at the right moment for each of both the Raspberry Pi and the laptop, the latter acting as the remote server.

In the next section, we will focus on the edge IoT functionality.

IoT foundation layer integration

This stratum is located in the bottom part of the software stack. The following figure shows that the **Middleware Layer** is built on top of it. It's Pusher API that will make the interface between both.

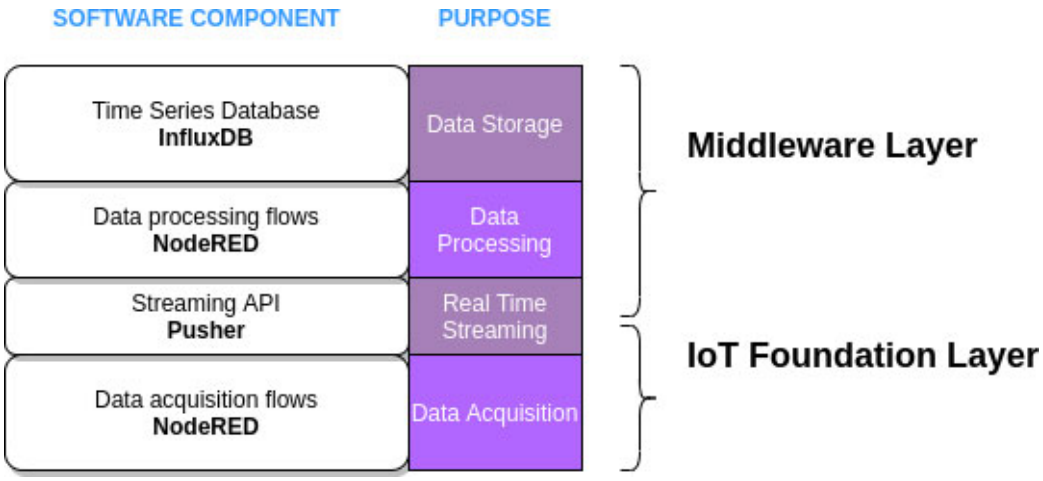


Figure 7.1: *Software stack for IoT Foundation and Middleware layers*

The **IoT Foundation Layer** covers two basic functionalities. These are as follows:

- to acquire environmental data from the Sense Hat
- to stream such data to Pusher app in the cloud

These two functionalities correspond to requirements 3.2 and 3.3, respectively, as defined in the previous chapter. The corresponding code has to run inside the Raspberry Pi, and it is derived from the software stack that we developed for the simulator of the Sense Hat.

In the framework of the application's architecture, the following figure provides an overview of where the code of this layer comes from in the book:

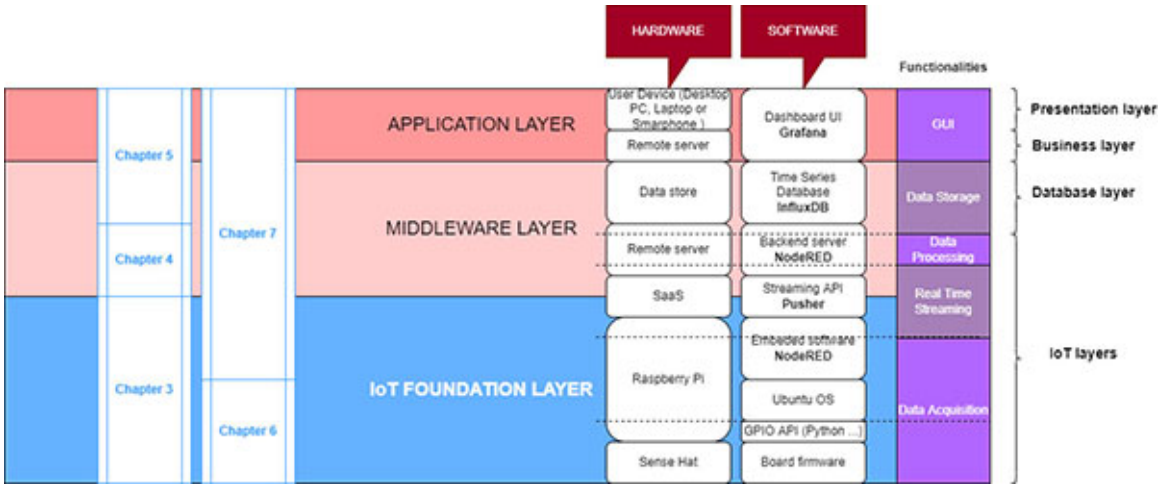


Figure 7.2: *Software coverage for the application per layer, component and chapter*

That is to say:

- *Chapter 6, The IoT Hardware Package* covers the functional integration of the Raspberry Pi and the Sense Hat.
- *Chapter 7, The IoT Software Package* covers the three software layers.
- *Chapters 2* to *5* constitute the learning path where every modular software component is introduced to be run and tested in your laptop.

This section covers requirements 3.2 and 3.3 as described in the section Application architecture covered in *Chapter 6 (The IoT hardware Package)*:

- *3.2 Environmental data acquisition from the Sense Hat using NodeRED*
- *3.3 Streaming data to Pusher app in the cloud*

Requirement 3.2 will be covered in the subsection Data acquisition with NodeRED, where the compliance will be demonstrated. As part of it

streaming to Pusher platform (that relates to requirement 3.3) will expose the environmental data to the outside world.

Before proceeding with the practical part, we need to split the set of NodeRED flows, since a group of them will reside in the Raspberry Pi, and the rest on the remote server. This is the topic of the next subsection.

Splitting the flows of the single NodeRED instance

As explained in section *Software architecture for the IoT application* of *Chapter 5, Storing and Graphing Data Streams with InfluxDB and Grafana*, the whole set of NodeRED flow is divided in two groups: one to be hosted on the IoT device, and the other on the remote server as shown in the following figure:

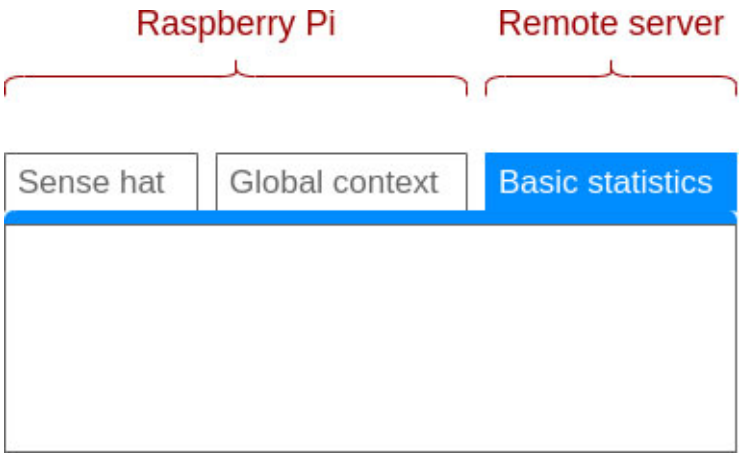


Figure 7.3: *Split of the flows between IoT device and remote server*

In that case we should include a test flow in the Raspberry Pi to make sure that the data is being sent to Pusher. That is the reason to add the tab named `Debug Pusher` in that NodeRED instance. So, let's proceed to deploy NodeRED on the Raspberry Pi using Balena.

Setting up the Balena application

This subsection reproduces what was carried out in the section *IoT hardware unit testing* of the previous chapter. We used simple software to show the compliance with the two requirements (recap them reading

the note below). Now, we will do something similar with the difference that the software that will run on the Raspberry Pi will be the one needed by the IoT application.

This subsection demonstrates compliance with requirements 6.3 The IoT device (Raspberry Pi) is properly managed from Balena cloud, and 3.1 Sensors readings are retrieved from the Sense Hat, as defined in the section Application architecture in *Chapter 6, The IoT Hardware Package*.

In the previous chapter, the compliance for these requirements was verified with several Python scripts that were written just to test these requirements. In that case, we put the focus on testing the hardware, however now the focus is shifted to the software itself.

Following the process outlined in the section *IoT hardware integration* of the previous chapter, we provide the step-by-step instructions in the following steps:

1. **Create a new application from the Balena dashboard**. We will name it Sense-IoT. Select the default device type, that would be the version of Raspberry Pi you are using (3 or 4). The following figure shows the set-up using a Raspberry Pi 4:

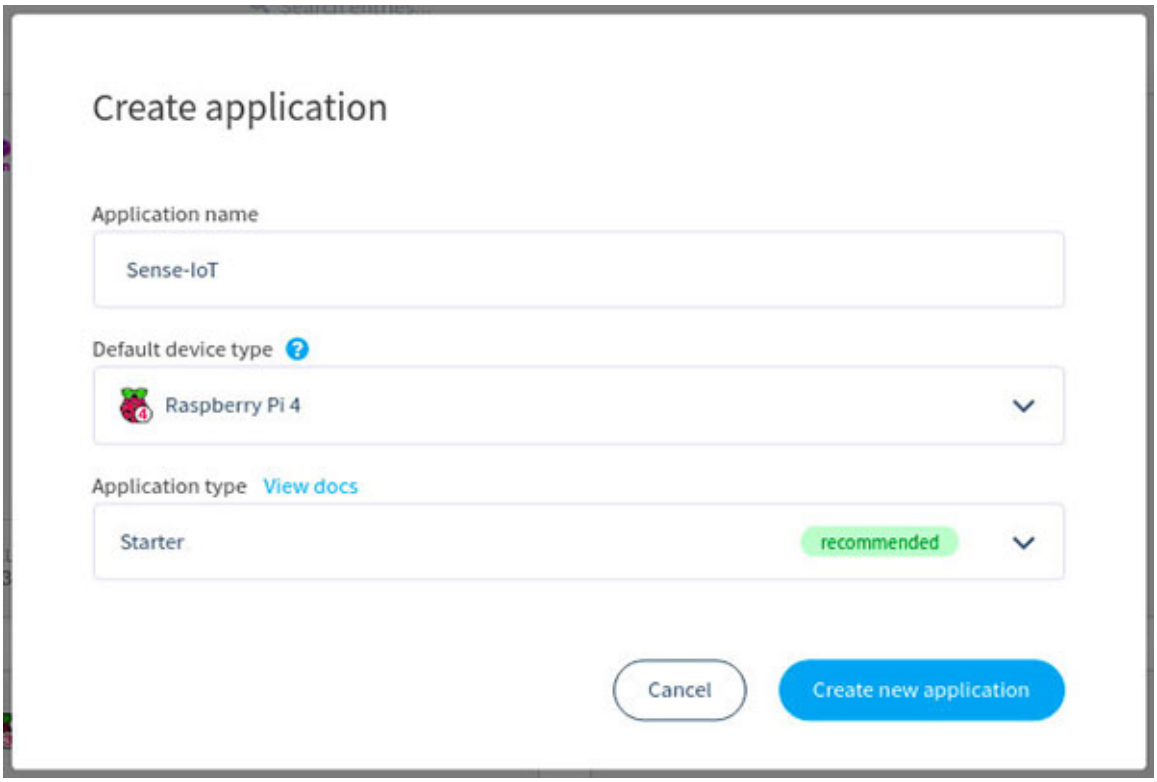


Figure 7.4: *Balena screen for setting up the IoT application with default device as Raspberry Pi 4*

2. Add your device by clicking the **`Add device`** button of the application window as shown in the following figure:

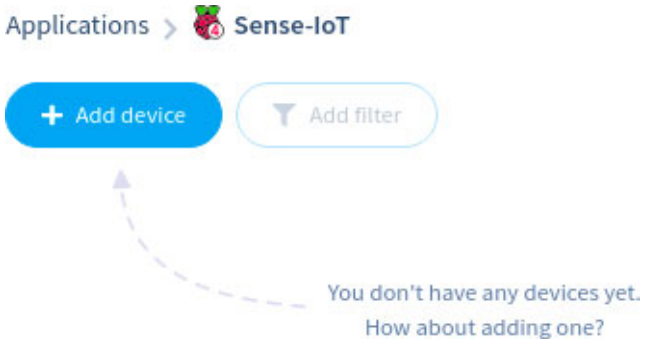


Figure 7.5: *Add a device to the Sense-IoT application*

3. **Specify Wi-Fi credentials,** and then download BalenaOS by pressing the button on the bottom-right part of the window as shown in the following figure:

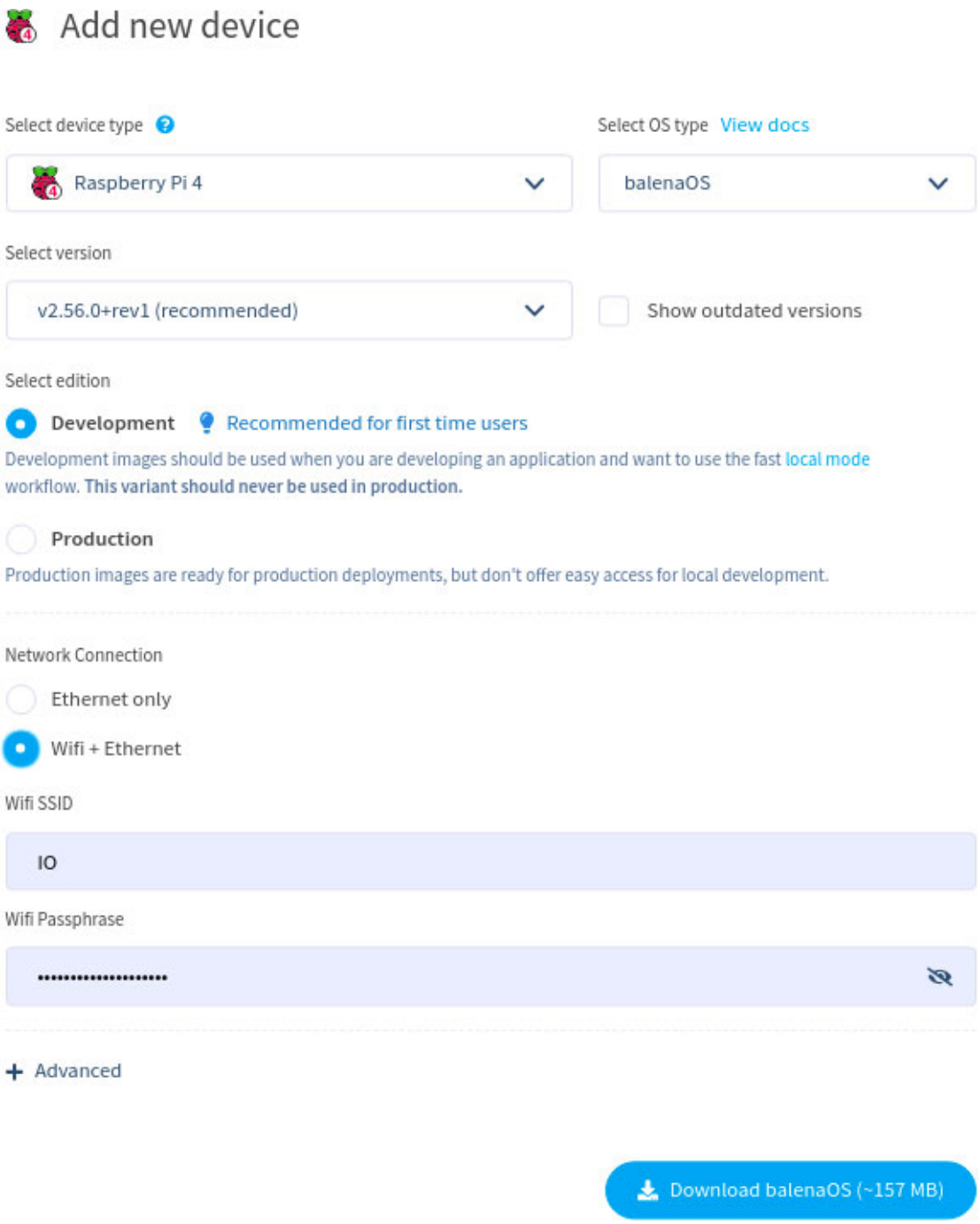


Figure 7.6: *Configuration window for the device to add to the Sense-IoT application*

4. Burn BalenaOS image to a micro-SD card using Etcher, an open-source tool available at **https://www.balena.io/etcher**.
5. Insert the micro-SD card into the slot of the Raspberry Pi ¾, and power it on. Remember that the Sense Hat board has to be plugged into the Raspberry Pi GPIO before powering it.

6. Wait until the new device appears in the Balena window of the application. This way, you can make sure that the device has been configured properly:

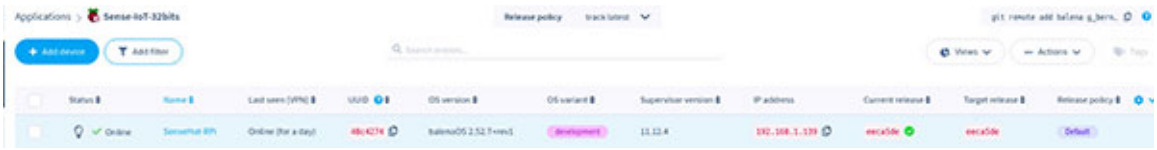


Figure 7.7: *Device is online in the window of the Balena application*

At this point, the Raspberry Pi is ready from the software point of view.

Data acquisition with NodeRED

This subsection demonstrates the compliance with requirement 3.2 Environmental data acquisition from the Sense Hat using NodeRED as stated in the section Application architecture in *Chapter 6, The IoT Hardware Package*.

In part 2 of the book, we developed and tested the NodeRED flows of the application with a simulator of the Sense Hat. Recap from *Chapter 5, Storing and Graphing Data Streams with InfluxDB and Grafana* that all the flows run in a single NodeRED instance.

Now, we should split them according to the flows of the single NodeRED instance as shown in *figure 7.3* above:

- One group will run on the NodeRED instance inside Raspberry Pi.
- The other group will be hosted on another NodeRED instance running on the remote server (i.e., your laptop for the practical purpose of the project).

The code that allows the deployment of the first group is contained in the repository at **https://github.com/Hands-on-IoT-Programming/chapter7-balena**, as explained in the *Technical requirements* section above.

Compared to the identical code in *Chapter 3, Data Acquisition and Real Time Streaming*, the NodeRED package `node-red-node-pi-sense-hat-simulator` is replaced by `node-red-node-pi-sense-hat`. That is, we now

use the module that reads environmental data from the actual hardware, instead of the simulated Sense Hat.

You can find the documentation for the package in the NodeRED documentation at **https://flows.nodered.org/node/node-red-node-pi-sense-hat**.

Following with the process outlined in the section IoT Hardware Integration of the previous chapter, let's complete the steps to deploy the code of the application on the Raspberry Pi:

- First of all, make sure you are logged into your Balena account using the Balena CLI command `balena login`. Then, choose the authentication method that you prefer.
- Clone the repository that contains the code to deploy:

```
$ cd ~/book_hands-on-iot
$ git clone https://github.com/Hands-on-IoT-
Programming/chapter7-balena.git
```

- Switch to the folder where the app stores the code and set authentication data, as well as the Pusher credentials:

```
$ cd ~/book_hands-on-iot/chapter7-balena/nodered/app
```

  Then, copy `.env.template` to a new file called `.env`. Open it, and specify passwords and credentials:

```
$ cp .env.template .env
$ nano .env
```

```
USERNAME=admin
PASSWORD='$2b$08$a3BqIzZBlX1ZZGymCPCeV.JNy0nEmyOIRdiN.hDjxXaHMzPk1Z/uO'
SECRETKEY=raspberry
DISABLE_NODERED_EDITOR=false
APP_ID=
APP_KEY=
APP_SECRET=
APP_CLUSTER=eu
```

Figure 7.8: *Specify NodeRED authentication data and Pusher credentials*

Part of the strings are blurred intentionally because it corresponds to credentials. This is what each variable provides:

- `USERNAME` and `PASSWORD` provide respectively the user and hash of the password for your NodeRED user. Password is raspberry in the example above.
- `SECRETKEY` supplies to the application a key to encrypt the credentials that are to be written in some of the flows, i.e., Pusher nodes.
- `DISABLE_NODERED_EDITOR`, if set to "true", the access to the NodeRED editor from the browser will be disabled (this is what is preferred when it is in production). For the purpose of development set its value to "false". When running in production, make sure to change it to "true" for providing a new security layer that will avoid any external editing of the flows.
- The rest of variables that begin with APP_, refer to the credentials of your Pusher application. This was explained in detail in section *Setup Pusher nodes to point to your account and app* of *Chapter 4, Real Time Data processing with NodeRED* (see *figure 4.1*).

The file flow7-balena_cred.json is the credentials complement of the file flow7-balena.json. That file is encrypted with the SECRETKEY that you supplied in the .env file. So that it works with the sample code of the repository, the SECRETKEY has to be equal to "raspberry". See the paragraph Managing Environment Variables below for a detailed description of how these credentials are passed to NodeRED.

Finally, deploy the code to the Balena application:

```
$ balena push Sense-IoT
```

`Dockerfile.template` inside `./nodered` folder corresponds to a 32 bits architecture compatible with Raspberry Pi 3 and 4. If you want to run on top of the 64 bits OS version, overwrite `Dockerfile.template` with the file `Dockerfile.template.64bits`:

```
$ cp Dockerfile.template.64bits Dockerfile.template
```

Then, run the same command above:

```
$ balena push Sense-IoT
```

This action launches the automated Balena pipeline that carries out the following operations:

- Push the code to Balena servers.
- Build the NodeRED application as a Docker container on Balena servers.
- Download the Docker image to all the devices in the application (only one for our particular case).
- Start the Docker container NodeRED in the device. In the scope of Balena, containers are known as services.

There is the alternative method of pushing the code to Balena, that consist of using plain git:

```
$ git remote add balena <BALENA_USER>@git.balena-cloud.com:
<BALENA_USER>/sense-iot.git
$ git push balena master
```

But this option should not be used because it will not push the `.env` file. The reason for that is that `.env` is listed inside the `.gitignore` file under the app folder inside the NodeRED directory.

This `.env` file is intentionally hidden so that NodeRED and Pusher credentials are not stored in GitHub servers. Otherwise, any user cloning our repository will have access to our secret information.

The following figure shows the starting process of the **`NodeRED`** service:

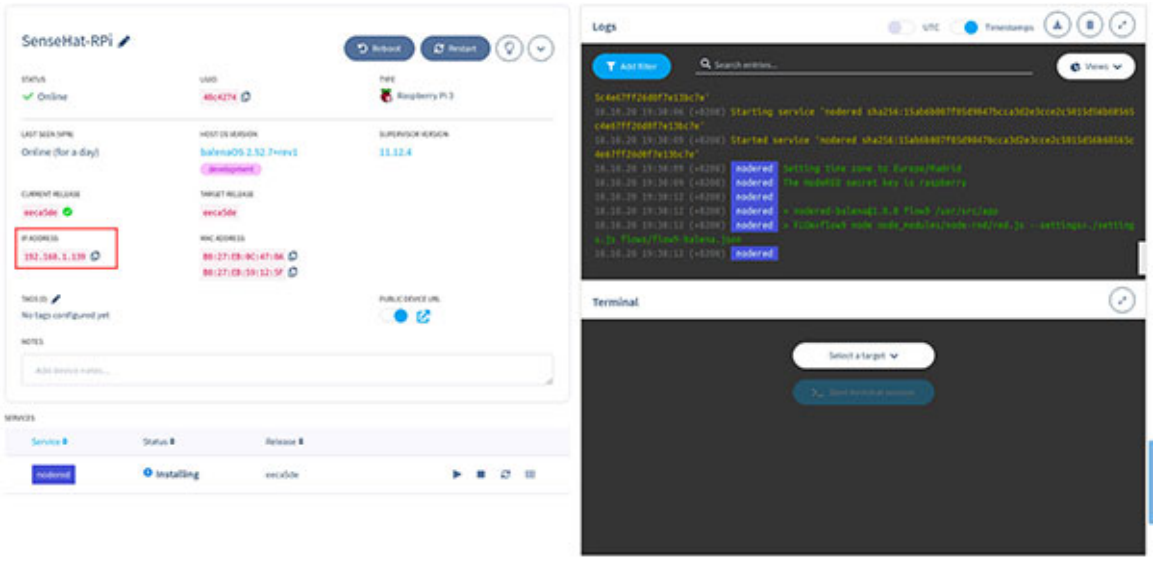


Figure 7.9: *Start nodered service in the device of the Sense-IoT Balena application*

The log window in the top-right part of the device dashboard shows what is happening in the Raspberry Pi. After several seconds you will see in the start-up information provided by NodeRED:

```
Welcome to Node-RED
===================

18 Oct 19:30:16 - [info] Node-RED version: v1.0.6
18 Oct 19:30:16 - [info] Node.js version: v12.18.3
18 Oct 19:30:16 - [info] Linux 4.19.118 arm LE
18 Oct 19:30:18 - [info] Loading palette nodes
18 Oct 19:30:23 - [info] Dashboard version 2.19.4 started at /ui
18 Oct 19:30:23 - [info] Settings file : /usr/src/app/settings.js
18 Oct 19:30:23 - [info] Context store : 'default'
[module=localfilesystem]
18 Oct 19:30:23 - [info] User directory : /usr/src/app/flows
18 Oct 19:30:23 - [warn] Projects disabled :
editorTheme.projects.enabled=false
18 Oct 19:30:23 - [info] Flows file : /usr/src/app/flows/flow7-
balena.json
18 Oct 19:30:23 - [info] Server now running at
http://127.0.0.1:80/
18 Oct 19:30:23 - [info] Starting flows
```

```
18 Oct 19:30:23 - [info] Started flows
```

The line marked in bold letters tells us how to access the NodeRED instance. It says it is running in port 80 of the Raspberry Pi:

```
http://127.0.0.1:80/
```

`127.0.0.1` is the IP of localhost for the Raspberry Pi. Since you want to access from your laptop, take note of the IP of the IoT device (see the rectangle that identifies it in the *figure 7.8* above, `192.168.1.139` in my case). Then, access to the NodeRED editor with the URL:

```
http://192.168.1.139/
```

Port 80 is the default, so you don't have to write it. Once the editor appears, the forefront tab Sense hat is the one that reads the sensors and pushes the data to Pusher.

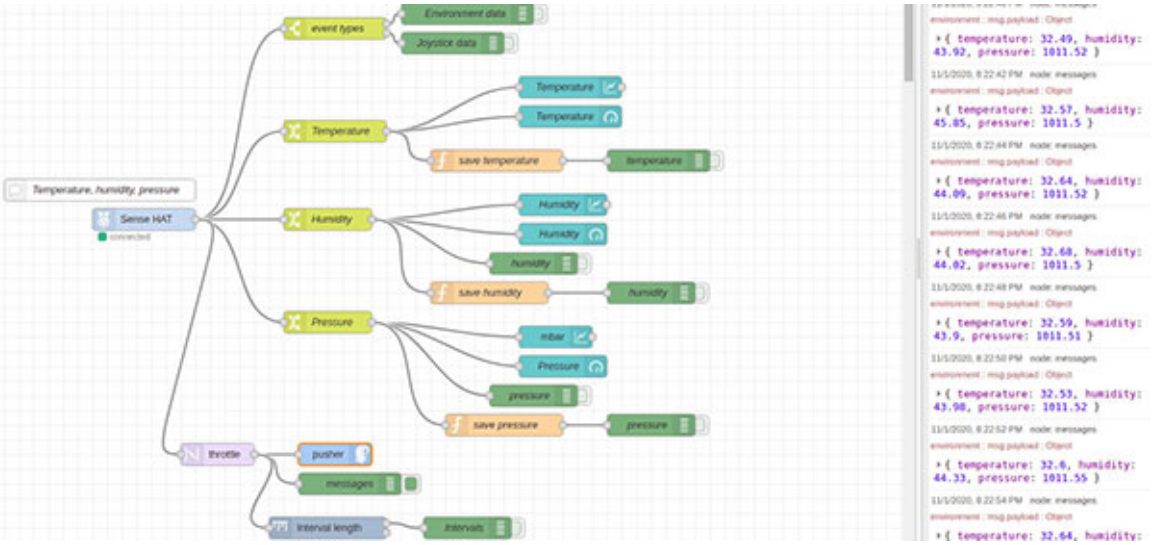


Figure 7.10: *NodeRED flows running in the Raspberry Pi: Sense hat tab*

In the section *Functional tests of the IoT Foundation Layer* below, we will check on our real set-up how the environmental data is being acquired.

In the next paragraph, we will briefly explain how environment variables are passed to NodeRED.

Managing environment variables

Visit the flow in the tab **ENV** and click on the button of the timestamp node as shown in the following figure:

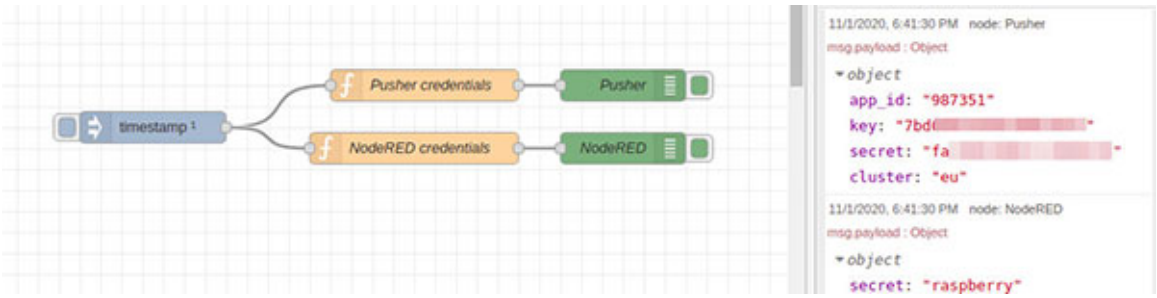


Figure 7.11: *Accessing Pusher and NodeRED credentials from a NodeRED flow*

If you double click on the function node Pusher credentials, you will see the code that is inside:

```
msg.payload = {
  app_id: env.get("APP_ID"),
  key: env.get("APP_KEY"),
  secret: env.get("APP_SECRET"),
  cluster: env.get("APP_CLUSTER")
}
return msg;
```

You can check that every environment variable you want to see is accessed with the function **`env.get("<variable_name>")`**. The same logic applies for the other node NodeRED credentials.

The function **`env.get`** is defined at the application level. For the app username and password we will show that they are accessed with the native node object **`process.env.<variable_name>`**. To do so, in the following code snippet below we have extracted the relevant lines of the settings file located at **`./nodered/app/settings.js`**

```
module.exports = {
  …
credentialSecret: process.env.SECRETKEY,
  …
adminAuth: {
   type: "credentials",
   users: [{
```

```
    username: process.env.USERNAME,
    password: process.env.PASSWORD,
    permissions: "*"
   }],
   default: {
    permissions: "read"
   }
  },
  …
  disableEditor: process.env.DISABLE_NODERED_EDITOR,
  …
}
```

You can verify that for variables **`SECRETKEY`**, **`USERNAME`**, **`PASSWORD`** and **`DISABLE_NODERED_EDITOR`** variables are prepended by **`process.env`**. This object contains all the environment variables in the scope of the application. You can retrieve them by making a copy of the Pusher credentials node and replacing its content by:

```
msg.payload = process.env
return msg;
```

```
When clicking on the timestamp node, this will be the output of
the new node:
{
 "username":"admin",
 "password":"$2b$08$a3BqIzZBlX1ZZGymCPCeV.JNy0nEmyOIRdiN.hDjxXaHM
 zPk1Z/uO",
 "secret":"raspberry"
}
```

The output is a JSON formatted object that contains the variables defined in **`setting.js`** through the **`process.env`** node object.

Functional tests of the IoT foundation layer

This section provides the compliance demonstration for the requirements on the IoT foundation layer:

- *Environmental data acquisition from the Sense Hat using NodeRED*
- *Streaming data to Pusher app in the cloud*

You can contextualize this section within the scope of the Verification Matrix as depicted in table 1 of *Chapter 6, The IoT Hardware Package*.

Acquiring data with NodeRED

For the first requirement, just visit the NodeRED instance running in the Raspberry Pi and look at the forefront flow, i.e., Sense hat. The following figure shows the debug tab in the right sidebar of the current flow.

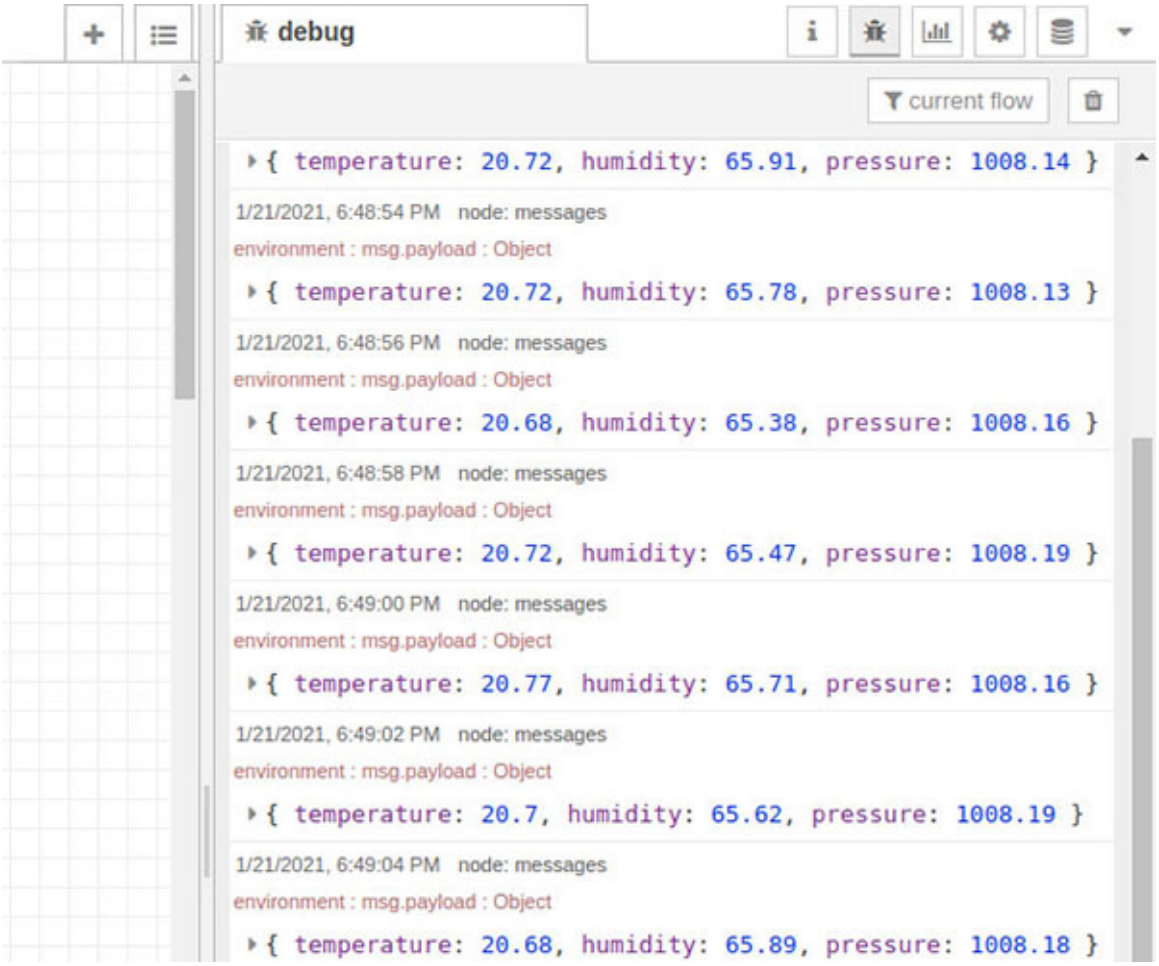


Figure 7.12: *Debug tab of the forefront flow Sense Hat*

You can appreciate how the sensors are read every two seconds and the values streamed to the debug window. This result provides the

verification for the requirement *3.2 Environmental data acquisition from the Sense Hat*.

Streaming data with Pusher app

Regarding the other requirement *3.3 Streaming data to Pusher app in the cloud*, have a look at the Debug Pusher tab:

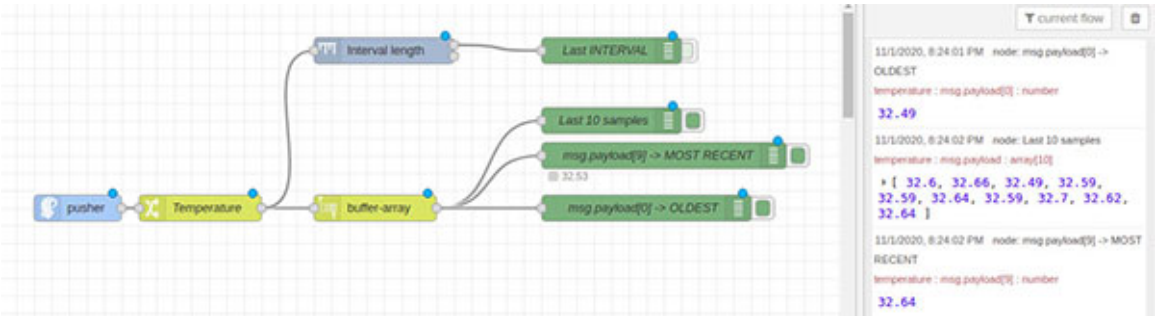


Figure 7.13: *NodeRED flows running in the Raspberry Pi: Debug Pusher tab*

Then, put the focus on that tab, reading a data flow as shown here:

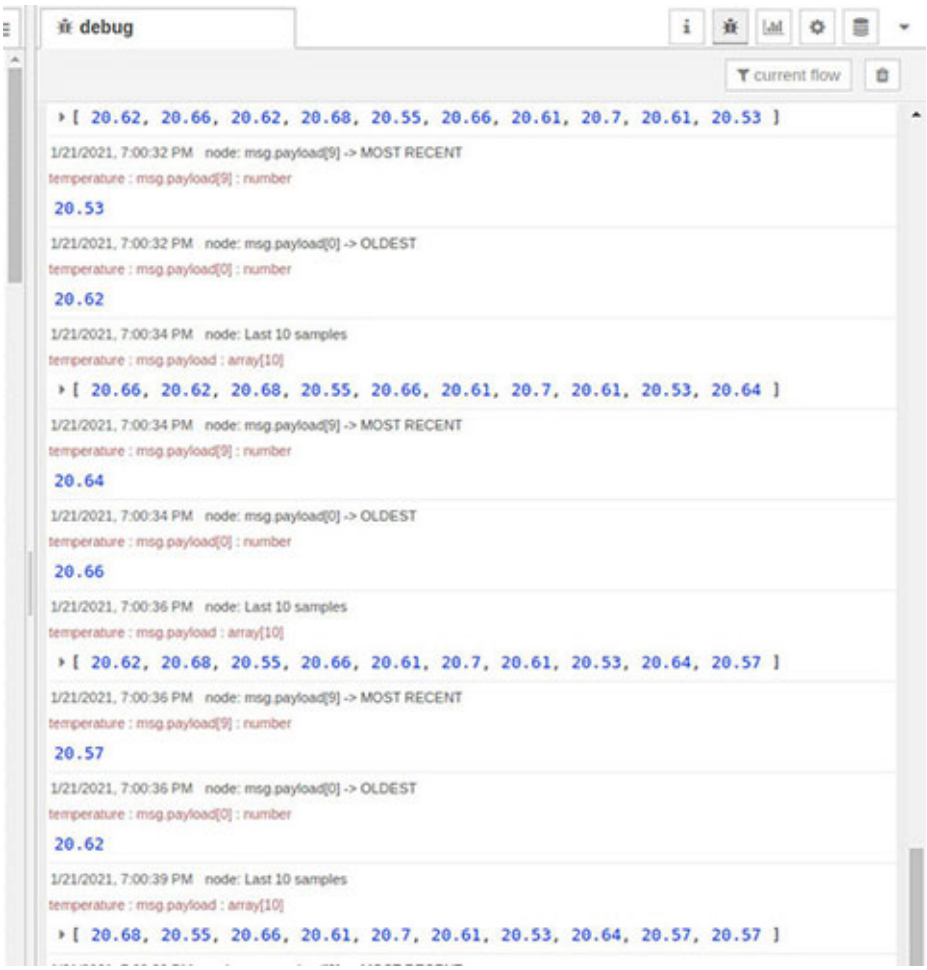


Figure 7.14: *Debug tab of the flow Debug Pusher*

According to the flow whose input origins in the pusher in node, we obtain the following outputs:

- array with the last 10 measurements of the temperature
- the most recent measurement, i.e., last element of the array
- the oldest measurement, i.e., first element of the array

You can see that a new point is added every two seconds, as dictated by the Sense hat flow. Although both flows are in the same NodeRED instance, the first one reads the sensor and streams the data to Pusher, while the second is connected to the Pusher channel and retrieves one point every two seconds.

This result provides the verification for the second requirement *3.3 Streaming data to Pusher app in the cloud*.

Once we have checked that the IoT foundation layer provides the expected functionality, we can go forward to the next layer, i.e., the middleware.

Middleware layer integration

This layer provides the connection between data from the devices and the end user interface. It also provides the functionality **Extract, Transfer and Load** (**ETL**) that transforms the raw data from the sensors into meaningful dashboards for the user. The following figure shows that central position between the IoT Foundation and middleware layers.

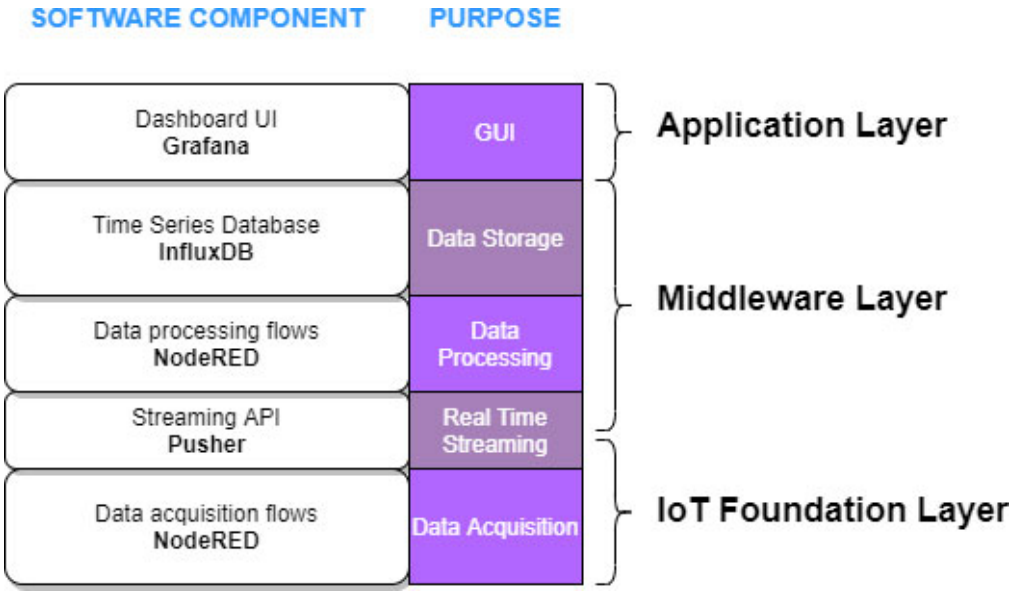


Figure 7.15: *Software stack for IoT Foundation, Middleware and Application layers*

The **Middleware Layer** covers three basic functionalities. They are as follows:

- to *read the data stream from Pusher* app
- to *compute basics statistics* of environmental data
- to *store the time series of environmental data* in database

These three functionalities correspond to the *requirements 4.1, 4.2, and 5.1*, respectively, as listed in *Chapter 6, The IoT Hardware Package* . The corresponding code has to run inside the laptop (emulating the

remote server), and it is derived from the software stack that we developed for the simulator of the Sense Hat in part 2 of the book.

Note: This section covers requirements 4.1, 4.2 and 5.1 as described in the section *Application architecture* from *Chapter 6, The IoT Hardware Package*:

- 4.1 Receiving the environmental data from Pusher app
- 4.2 Obtaining basic statistics of data in the laptop
- 5.1 Storing the environmental data in the InfluxDB database

The requirement 4.1 will be developed in section Middleware Layer Integration, paragraph Retrieving data from Pusher. The same applies to requirement 4.2 in the following paragraph Performing statistics calculations, and for 5.3 in the last paragraph storing historical with InfluxDB.

In the diagram layer-code matching, you can see what chapter provides the code for each component in this layer:

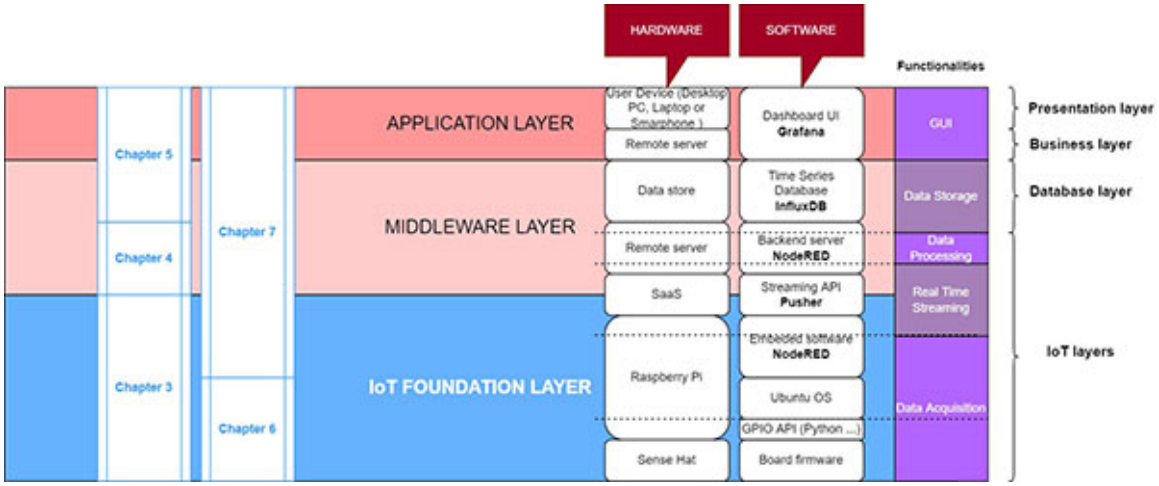


Figure 7.16: *Diagram layer-code matching: Middleware Layer components are covered in this chapter*

From the preceding figure, it can be checked that the pieces of code for the middleware layer components come from the following:

- *Chapter 4, Real Time Data Processing with NodeRED*

- *Chapter 5, Storing and Graphing Data Streams with InfluxDB and Grafana*

At this point, you should have an overall understanding of the context, so let's proceed with the implementation.

Retrieving data from Pusher

First of all, you should recap the division of NodeRED flows explained in paragraph *Splitting the flows of the single NodeRED instance* (see *figure 7.3* above):

- The first group covers the data acquisition and streaming to Pusher from the Raspberry Pi. This corresponds to the implementation we made in the section *IoT Foundation Layer integration* above.
- The second group implements the data processing in real time, and this functionality happens in the remote server. The input data is read from the Pusher stream. Hence, this group covers what needs to be done in the current section, i.e., *Middleware Layer integration*.

The code that allows the deployment of this second group is contained in the repository at **https://github.com/Hands-on-IoT-Programming/chapter7**, as detailed in the *Technical requirements* section above.

This subsection demonstrates the compliance with requirement 4.1 Receiving the environmental data from Pusher app as stated in the section Application architecture from *chapter 6 (The IoT hardware package)*.

First, clone the GitHub repository, then install dependencies as follows:

```
$ $ cd ~/book_hands-on-iot
$ git clone https://github.com/Hands-on-IoT-
Programming/chapter7.git
$ cd ~/book_hands-on-iot/chapter7/nodered/app
$ cd ./nodered
```

```
$ npm install
```

Then copy `.env.template` to a new file called `.env`. Open it and specify your own password and Pusher credentials:

```
$ cp .env.template .env
$ nano .env
```

```
USERNAME=admin
PASSWORD='$2b$08$a3BqIzZBlX1ZZGymCPCeV.JNy0nEmyOIRdiN.hDjxXaHMzPk1Z/uO'
SECRETKEY=raspberry
DISABLE_NODERED_EDITOR=false
APP_KEY=
APP_CLUSTER=eu
```

Figure 7.17: *Specify NodeRED authentication data and Pusher credentials in the remote server*

The explanation of the environment variables is the same provided in paragraph *Data acquisition with NodeRED* above, with the exception that in the case of Pusher you only need the `APP_KEY` credential (you are only reading from the channel).

After finishing the configuration, we can launch the remote NodeRED instance with the following script:

```
$ bash start.sh
```

Why don't we deploy the application with `npm run <SCRIPT>` like we did in past chapters? Let's look to the content of `start.sh` to understand why:

```
#!/bin/bash
# Source ENV variables from .env file
set -o allexport; source .env; set +o allexport
echo 'The NodeRED admin user is' $USERNAME
echo 'The NodeRED admin password hash is' $PASSWORD
echo 'The NodeRED secret key (for flow encryption) is' $SECRETKEY
# Start the app
npm run flow7
```

The key is in the third line, where three consecutive commands are executed:

- **`set -o allexport`**: Enables all following variable definitions to be exported.
- **`source .env`**: The **`.env`** file contains the key-value pairs, where key is the name of the variable and value is the content we want to be stored in that variable.
- **`set +o allexport`**: Disables the exporting feature as we do not need it anymore.

Afterwards, you can safely launch the application. Since the credentials for logging into NodeRED are loaded as environment variables and will be read from **`settings.js`**. This point was covered in paragraph Managing environment variables above.

The application execution is done with the last command of **`start.sh`**, i.e., the **`npm script npm run flow7`**. This will load NodeRED including the flow for this chapter, located in the path **`./nodered/flows/flow7.json`** of the repository.

For the NodeRED instance in the Raspberry Pi, the execution process is exactly the same. The only difference is that the command:

```
$ bash start
```

corresponds to the entry point of the Docker container that is deployed in the Balena platform. To check this, find the last line in the file `/nodered/Dockerfile.template of the Raspberry Pi repository chapter7-balena`:

```
CMD ["bash", "/usr/src/app/start.sh"]
```

This final line tells Docker to execute (CMD) in the container the command `"bash" with argument "/usr/src/app/start.sh"`

On launching NodeRED, you should see the log messages in the console showing you if the application is running properly.

In the next section, we will provide the evidence to check the requirements on middleware layer.

Performing statistics calculations

The integration of this part of the code is as described in section *Basic statistics* of environmental conditions of *Chapter 4, Real-time data processing with NodeRED*. There is no difference with respect to the integration of the final application.

The NodeRED flow Basic statistics is fed through the same Pusher in node. Whatever comes from Pusher stream- data from the simulator (in *Chapter 4, Real-Time Data Processing with NodeRED*) or from the physical Sense Hat (this chapter)-, does not matter from the point of the calculations. Just that now the input values are the actual values of temperature.

In the corresponding paragraph of section *Functional tests* of the middleware below, we will review that NodeRED flow.

Storing historical data in InfluxDB

The how-to of this feature was covered in section *Storing time series* of environmental conditions of *Chapter 5, Storing and graphing Data Streams with InfluxDB and Grafana*. You only need to check that Sense Hat data is being written to the proper InfluxDB measurement.

To check this point, access the flow in the tab `Store aggregated data`, and locate the influxdb out nodes. You should find that instant values are written to `sense_hat` measurement (*figure 7.16*) in the following figure:

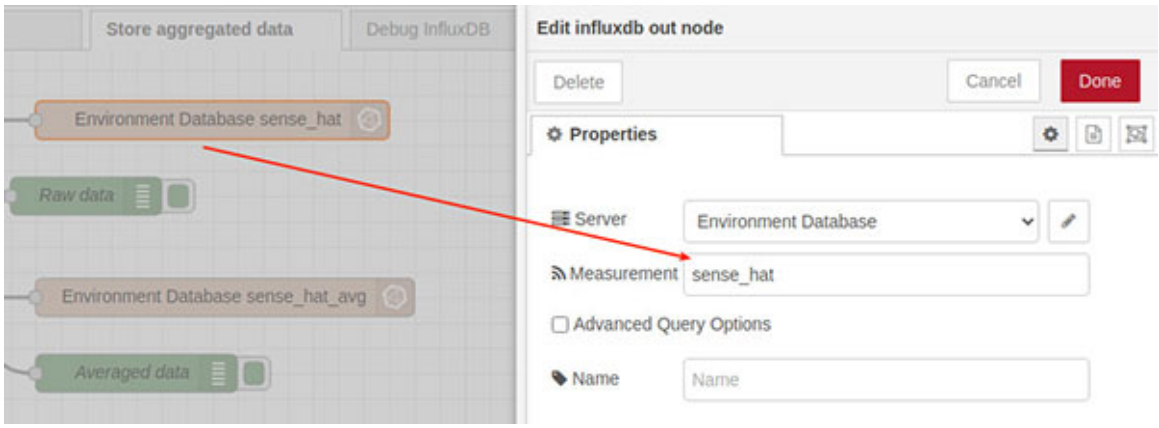


Figure 7.18: *Instant temperature values are being written to sense_hat measurement*

The same for the average temperatures that are written to `sense_hat_avg` measurement as shown in the figure:

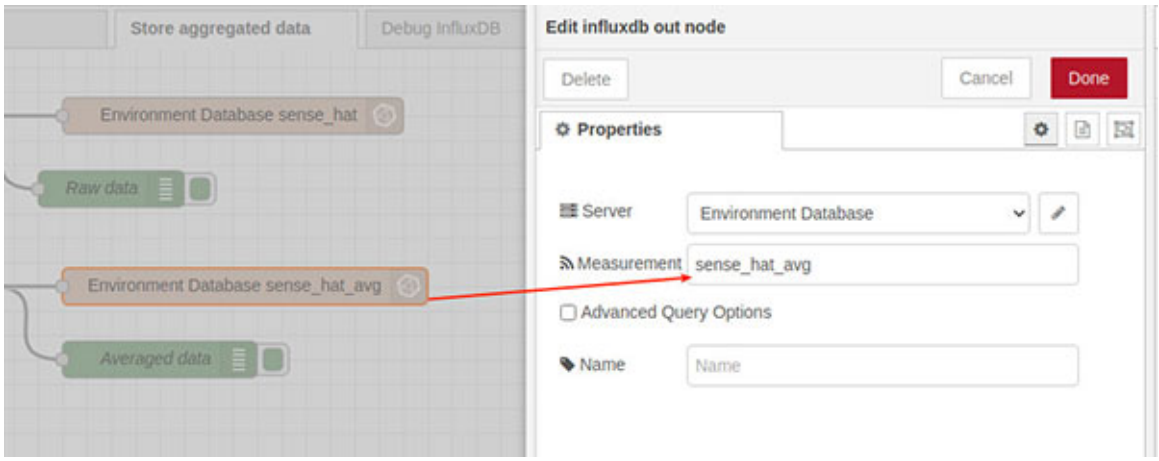


Figure 7.19: *Averaged temperatures are being written to sense_hat_avg measurement*

Once all the software is properly set in place, it's time to check the compliance with the requirements.

Functional tests of the middleware layer

This section provides the compliance demonstration for the requirements on the middleware layer as follows:

- *4.1 Receiving the environmental data from Pusher app*
- *4.2 Obtaining basic statistics of data in the laptop*
- *5.1 Storing the environmental data in the InfluxDB database*

You can contextualize this section within the scope of the *Verification Matrix* as depicted in table 1 of *Chapter 6, The IoT Hardware Package*.

After gathering positive results, you will make sure that the remote server works properly.

Receiving data and making basic statistics

First, let's recap the main flow in the `basic statistics` tab. We can enclose in rectangles the functional sub-flows as shown in the following image:

1. data acquisition verification
2. basic statistics of temperature
3. two wires that pipe temperature data (instant and average) to the storage flow

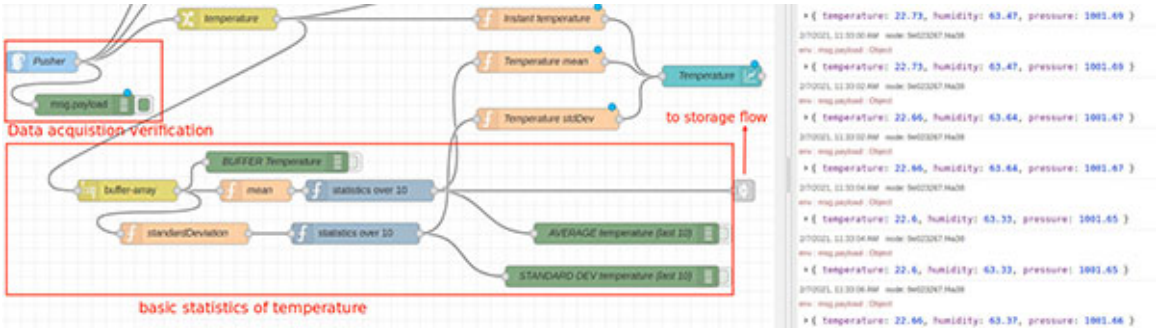


Figure 7.20: *Dissection of the main middleware flow. Debug log shows compliance with 4.1 Receiving the environmental data from Pusher app*

Based on that, we can make the following considerations:

- `Pusher in` node (leftmost part of the image) accounts for the first requirement, i.e., retrieving data from Pusher, in which each data point contains the set of three values temperature, humidity and pressure. You can see the Debug log window in the rightmost part showing the data received at every timestamp. This proves requirement 4.1 Receiving the environmental data from Pusher app.
- The nodes at the bottom—labeled as basic statistics of temperature—implement the real-time processing of the data. See the log in the right Debug window, shown in *figure 7.19* below, where the results of the statistical calculations are logged in real time.

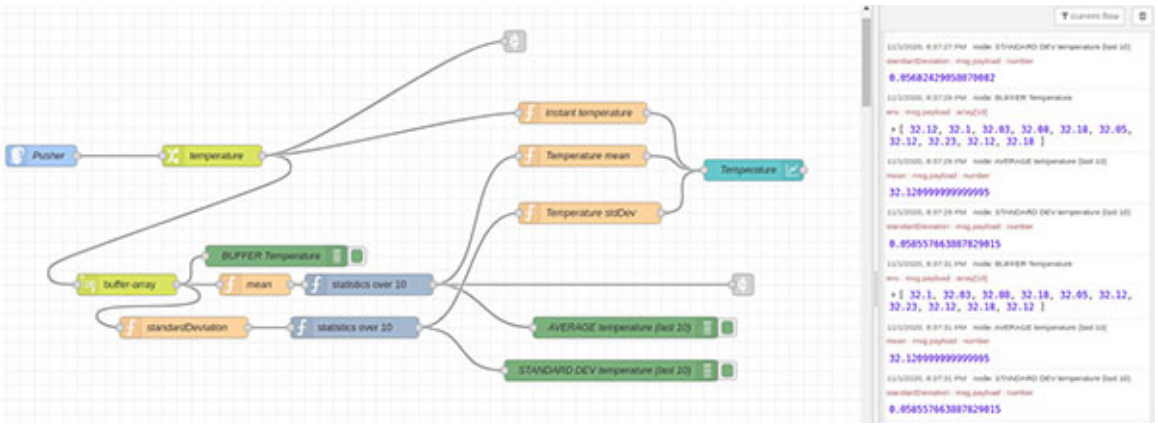


Figure 7.21: *Averaging the last 10 measurements and computing their standard deviation(Debug log shows compliance with 4.2 Obtaining basic statistics of data in the laptop.)*

The upper set of nodes linked to the chart node provides the graphical representation of such data as shown in the following figure:

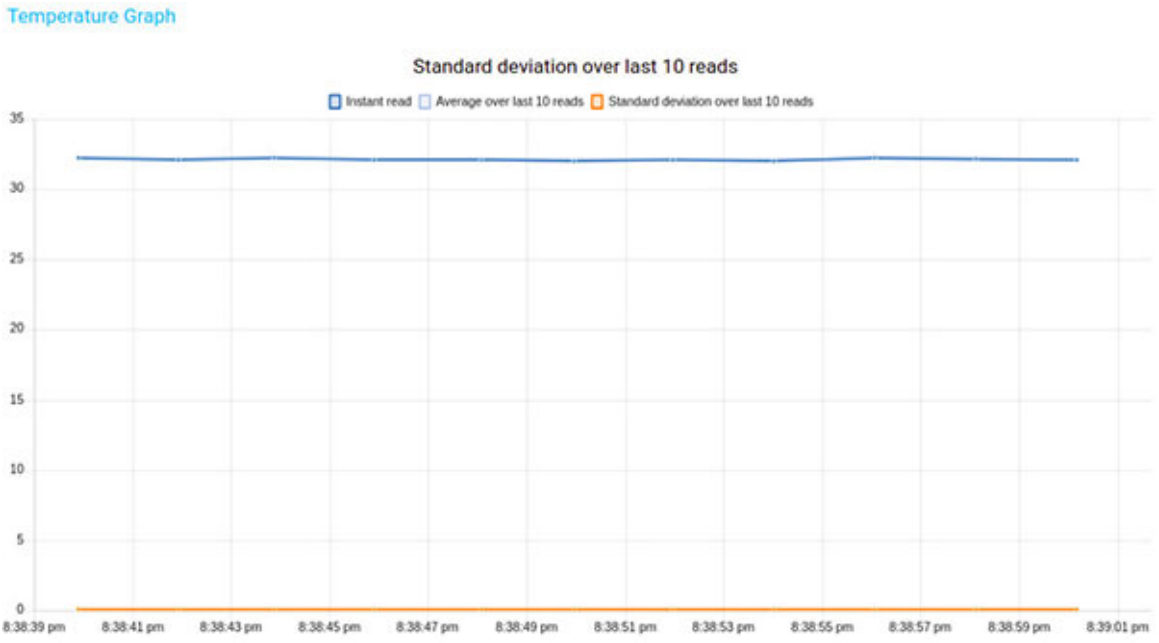


Figure 7.22: *Graphical representation of the temperature time series: instant value, average and standard deviation over the last 10 measurements*

Having all this in mind, we can move forward to the next requirement, *4.2 Obtaining basic statistics of data in the laptop*.

Look at the log in the Debug window, shown in the *figure 7.19* and the real time chart in *figure 7.20*. There, you can find the real-time stream of:

- the last ten measurement
- their average
- the standard deviation

You can check that the standard deviation is almost zero all the time. This is due to the fact that for ten measurements in a 20 seconds interval, we do not expect any actual temperature variation. Furthermore, what this value shows is the noise of the temperature sensor embedded in the Sense Hat. As a complementary exercise, let's quantify that value:

- The standard deviation is about *0.05 ºC*. Since the environment temperature is *20ºC*, this means that the noise is *0.05/20x100 = 2.5%*, a very acceptable value for such a low-cost sensor.

Finally, we will check that the data points are properly stored in the database. This is in the scope of the verification of the last requirement *—5.1 Storing the environmental data in the InfluxDB database—* covered in the next paragraph.

Checking the storage of the historical data

The NodeRED flows in *figure 7.21* below are hosted in the tab `Store aggregated data`, and they carry out the insertion of the temperature values in our InfluxDB database.

The logic of how they accomplish it was covered in section *Storing time series* of environmental conditions of *Chapter 5, Storing and Graphing Data Streams with InfluxDB and Grafana*, and how to configure the Influxdb nodes was explained in paragraph Storing historical data in InfluxDB above.

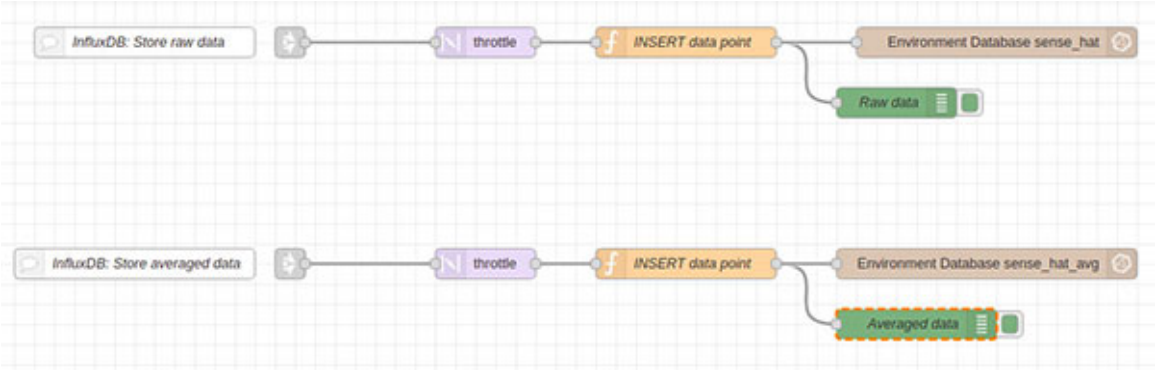


Figure 7.23: *Pushing instant and average temperatures to InfluxDB*

To show the compliance with requirement *5.1 Storing the environmental data in InfluxDB*, we just need to read the temperature measurements from the database. To do so, there is an additional tab called `Debug InfluxDB`, whose configuration is shown in the following figure.

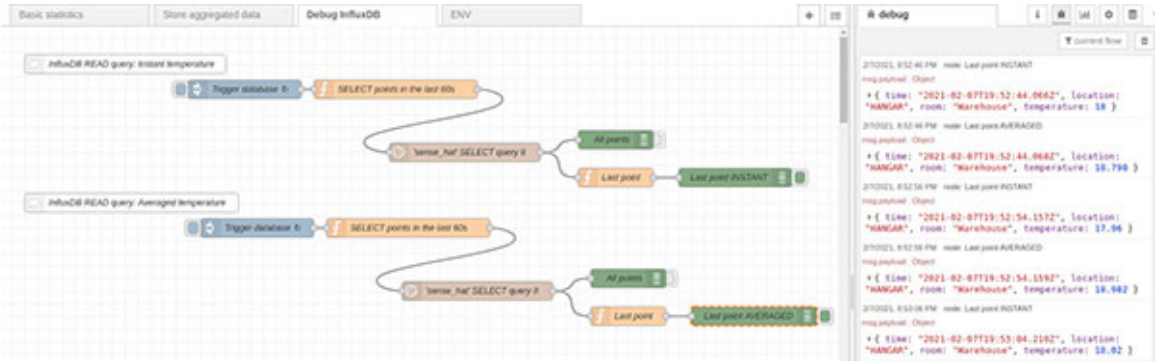


Figure 7.24: *NodeRED tab to check that data points are being written to the database*

We are reading two measurements, `sense_hat` for the instant temperature and `sense_hat_avg` for the average, and their values are sent to the `debug` tab of the sidebar.

So far, we have completed the middleware layer. In the next section, we will cover the same topics for the application layer.

Application Layer Integration

This layer provides the user interface that displays the temperature time series stored in the database. As such, it should provide a good user experience, which makes Grafana observability platform, the open-

source tool that sits on top of the software stack (recap *figure 7.14* above). Hence, as covered in paragraph *The Application Layer* of *Chapter 6, The IoT Hardware Package*, the requirements to comply with this functionality are:

- *5.2 Grafana connecting to the InfluxDB database*
- *5.3 Showing the Grafana dashboard via the web browser*

We will configure a Grafana dashboard for visualization data from the actual Sense Hat in the following section. Afterward, we will pass the tests to show compliance with the requirements listed above.

Steps to create the dashboard of the Sense Hat

The procedure is quite similar to what is described in paragraph *Creation of graph for the Sense Hat simulator* above. Everything is exactly the same up to Step 9, where we are presented with the new panel configuration page (step 10). The only thing you have to change is the measurement. Remember that Sense Hat data are stored in measurement `sense_hat` of the environment database. So, you have to select the field temperature in the box below. See the following figure to localize the parameters that you have to select:

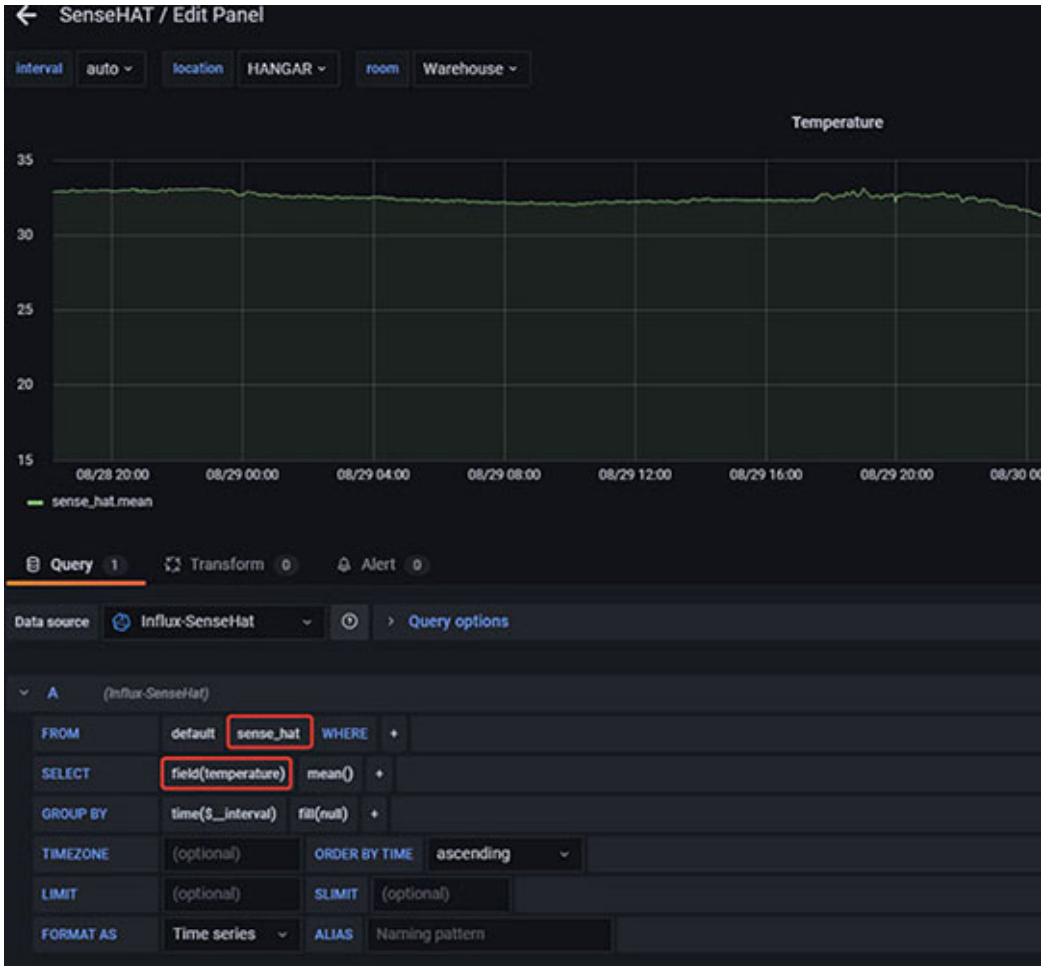


Figure 7.25: *Configuration of the panel for the temperature graph*

Then, save the panel and you should see something like the graph in the following figure:

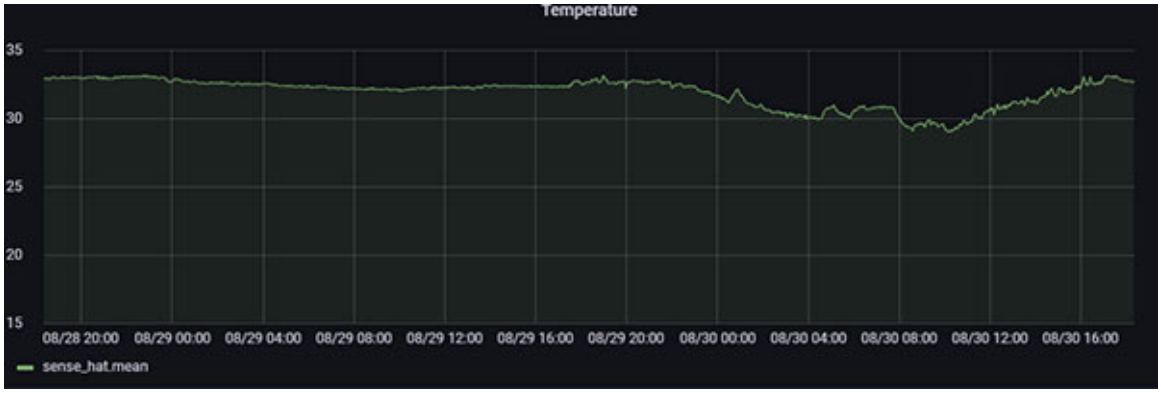


Figure 7.26: *Time series of temperature from 28 to 30 August 2021*

Functional tests of the Application Layer

Having completed the steps above, and visualizing the graph you have proved the requirements of this layer:

- *5.2 Grafana connecting to the InfluxDB database*
- *5.3 Showing the Grafana dashboard via the web browser*

Congratulations! Your IoT application is completed, and you have a nice interface to follow the time evolution of the Sense Hat temperature.

Conclusion

In this chapter, we focused on the IoT software of the project. Taking the modular code developed in *chapters 3*, *4* and *5*, we made slight changes to work with the physical Sense Hat.

Connecting the software modules of the different functionalities, you have understood how to practically use APIs to wire pieces of software that can work together if they agree on the interface between them.

In addition, following the system engineering approach explained in the previous chapter- where the requirements for every function of the software were defined- we have explained how to verify functionalities in the three layers: *Foundation*, *Middleware*, and *Application*.

Hence, you have completed the development cycle of a software project and should have acquired the skills to set requirements and select verification methods for other applications that you may develop in the future.

Points to remember

- *IoT Foundation Layer* solved the edge IoT part of the project, acquiring data in the physical location and pushing it to the cloud for processing on a remote server.
- *Middleware Layer* transforms raw sensors' readings (raw) into processed data.
- This second layer also provides the data persistence feature to track the history of environmental conditions.

- Based on that, the *Application Layer* will offer the visual dashboard to the end-users to interpret data, allowing them to take action if some anomaly is found (for example, overheating the Raspberry Pi).

Multiple choice questions

1. **What layer is the Pusher API located in?**
    a. in the Middleware Layer
    b. in the Foundation Layer
    c. the interface between both of them
    d. in the Application Layer
2. **What should you do if a functional requirement is not met?**
    a. Going through every software component related to that requirement, identify the failing component, and fix the issue.
    b. Review the code of the application.
    c. Clone the code from the repository again and re-run the software, because probably you modified something unintentionally.
    d. Ask an expert for advice .
3. **Why do you need to set up environment variables for the software components?**
    a. because in that way a software component can be easily changed by other equivalent module
    b. up to you, not a key point of the application
    c. because it is a good practice to build configurable applications
    d. because it's easier than hard coding the values in the software
4. **Could you keep all the NodeRED flows in the Raspberry Pi, even when the application keeps on working?**

a. Yes, you can in this simple case because Raspberry Pi has enough computing capacity for making statistics of data
b. Not really, because in that case you would not have any machine where to install the database
c. Well, you could, but you should refactor most of the code to make it work with this configuration
d. The foundation and Middleware Layers would work, but not the Application layer

5. **What's the core foundation of a software project?**
    a. the tools you choose for developing the application
    b. the software architecture
    c. the software itself
    d. the expertise of the people who will make the development

Answers

1. **c**
2. **a**
3. **c**
4. **a**
5. **b**

Questions

1. What are the functional requirements of a software component?
2. What are the technical reasons to select InfluxDB as the database for the application? (There are a bunch of open-source databases out there…)
3. What requirements could be added to the Middleware Layer so that it is resilient (for example, a crash in the remote server that would make the application unavailable)?

Key terms

- **IoT software**: It carries data acquisition in the physical location.
- **Remote software**: It processes the raw data to produce statistics and provide data persistence in a database.
- **Software requirement**: It is a statement of the characteristics that the code of an application, or part of that code, has to comply with.
- **Functional requirement**: It is a statement of what the application has to perform at system level, i.e., integrated set of modules that compose it.

Index

A

B

C

D

E

F

G

I

L

M

N

P

R

S

T

U

V