TOOLS AND FEATURES

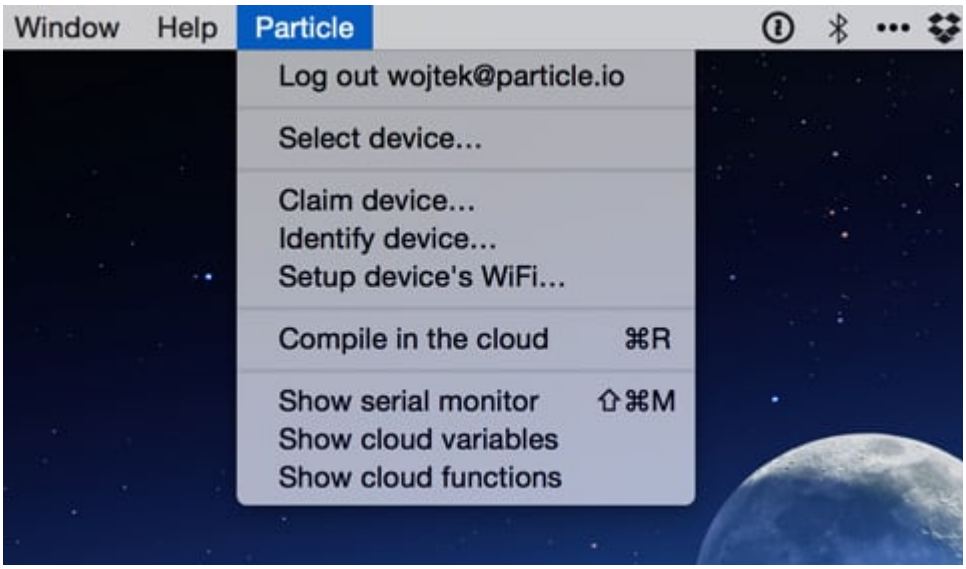
This section of the guide will give you an overview of tools and features that you may need to use the Particle Platform.

We've ordered the sections in order of what we think will be most useful to the average user, but feel free to browse them in whatever order you'd like.

If this is your first time on the guide, we recommend that you browse through the Getting Started section to connect and learn about your device, then use this section to check out some of the tools that might be useful to you as you build and develop. When you're thinking about scaling, head over to How to Build a Product with Particle for advice on taking your breadboard prototype to a scaled product.

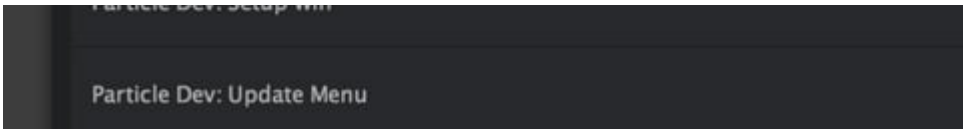
DESKTOP IDE (DEV)

Getting Started



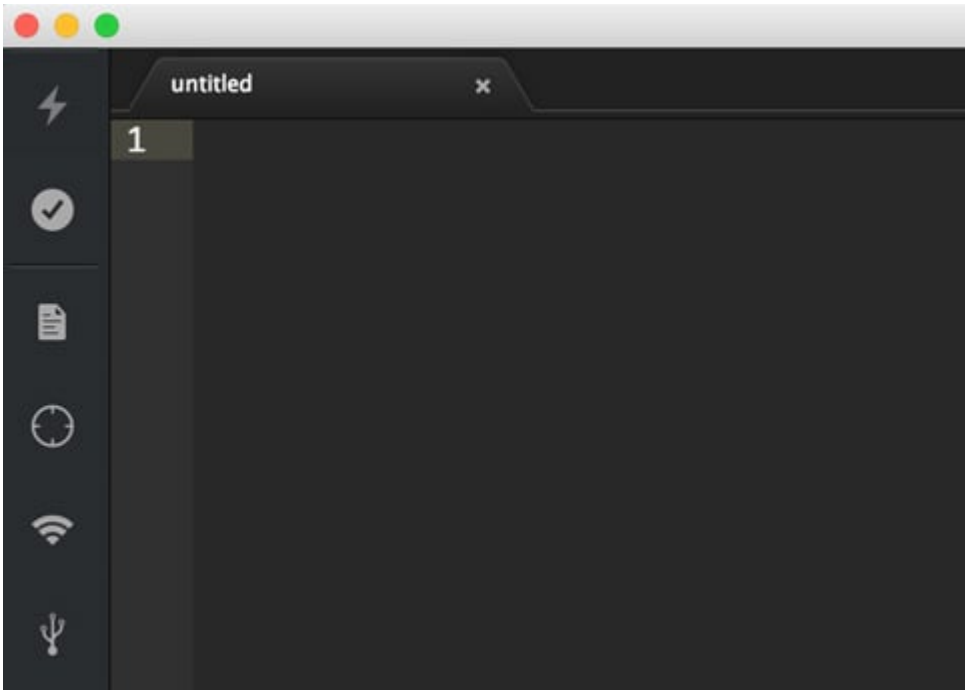
Particle Dev is a desktop application that allows you to work with local copies of your firmware files. However, **internet** access is required as the files are pushed to the Particle Cloud for compilation and returns a binary. i.e. This is not an offline development tool, yet.

All the commands are available from the **Particle** menu. The menu changes depending on whether you're logged in or have selected a device, so some of the commands will only show up once you're in the right context.



If you prefer a keyboard-oriented workflow, there's **Command Palette** with all available commands in a searchable list.

To show the palette press `Command` + `Shift` + `P` keys together on a Mac or `Control` + `Shift` + `P` on Windows.



Tip: you can change toolbar's position in settings.

There's also a toolbar on left side of IDE which contains shortcuts to the most frequently used commands like compiling and flashing (looks a lot like the one from Web IDE (Build), doesn't it?).

Logging In

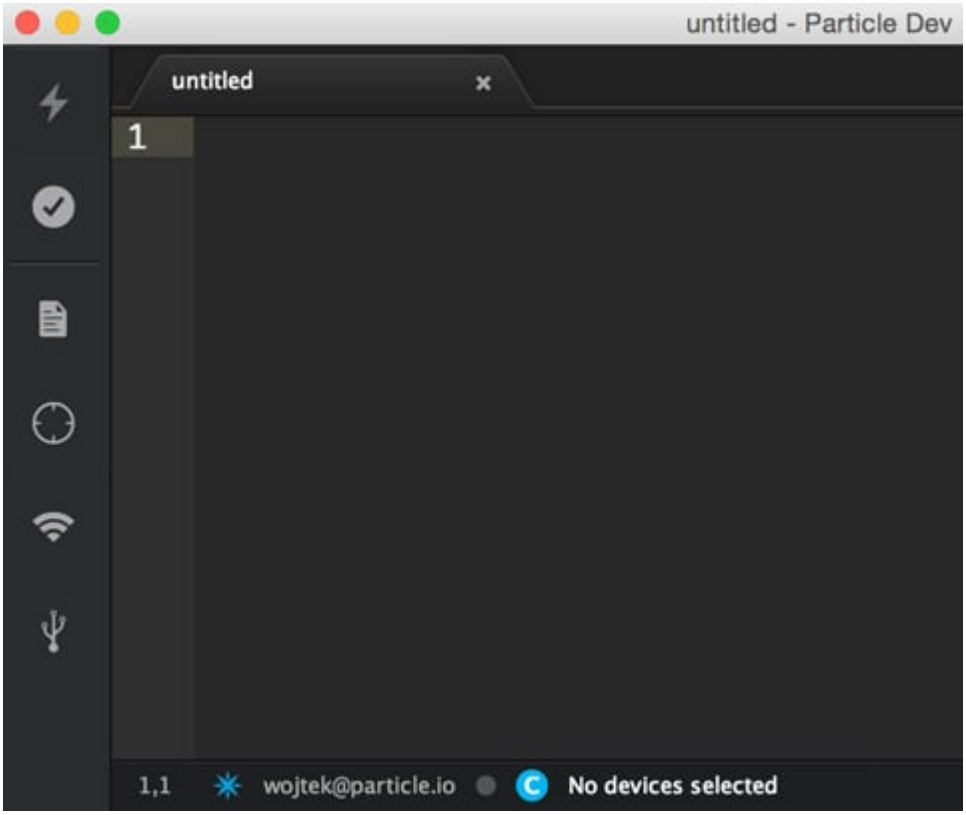
If you want to work on more advanced projects, Particle Dev could be the choice for you. Head over and download latest release:

Download for Windows >

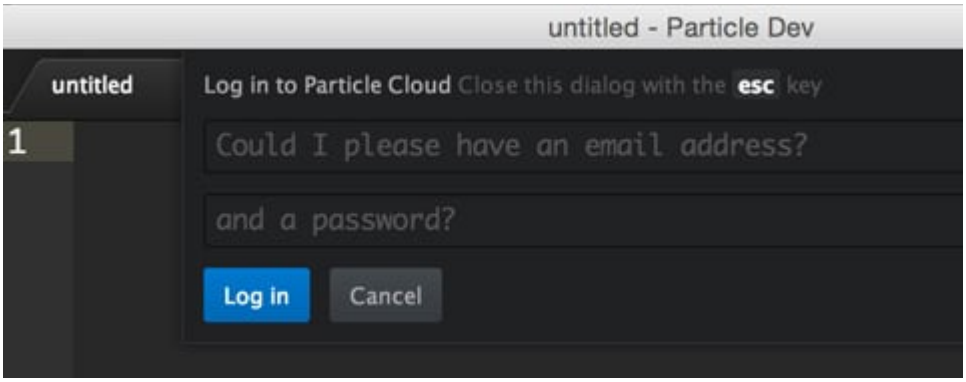
Download for Windows X64 >

Download for Mac >

Download for Linux >



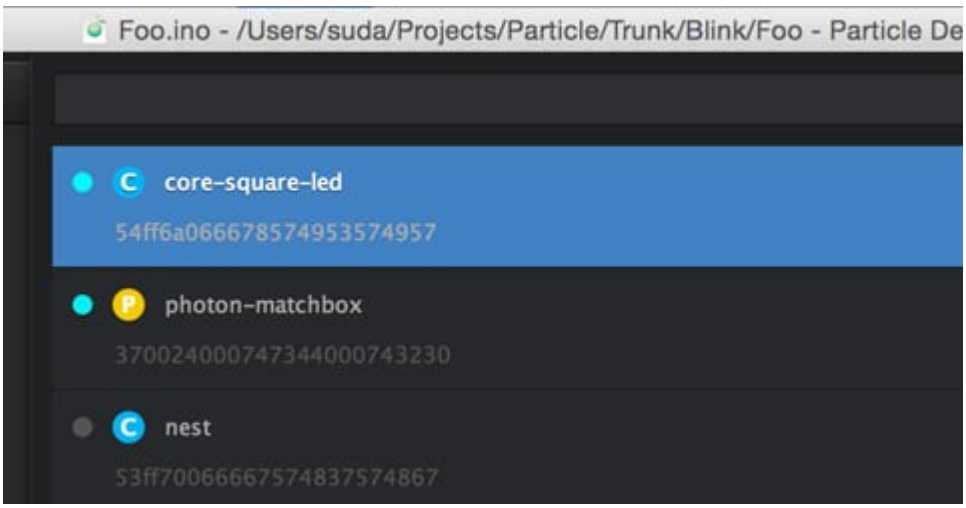
To access most of features you need to log in using your Particle account (which you can create here) by clicking the link on the bottom of the window.



Enter your email and password then click the "Log In" button. After a successful login, the dialog will hide and a link will appear at the bottom showing your current account email.

NOTE: When using Command Line you'll notice that log-in status is shared between Particle Dev and CLI. So if you successfully ran `particle login`, you will be logged in within the Particle Dev.

Selecting Device



Most features like **Flashing** or accessing **Cloud variables and functions** require selecting a target device they will interact with.

There are three ways to select core:

- Click **Select device** button in the left toolbar
- Click device's name on the bottom of the window
- Click on **Particle** -> **Select device...** menu

Then you will see list of all your devices along with an indicator of online status and platform. You can search for a specific one by typing its name. Clicking on the device or pressing `Enter` when a device is selected will select it.

Compiling Code

Before compiling your project, make sure your project files are in a dedicated directory.

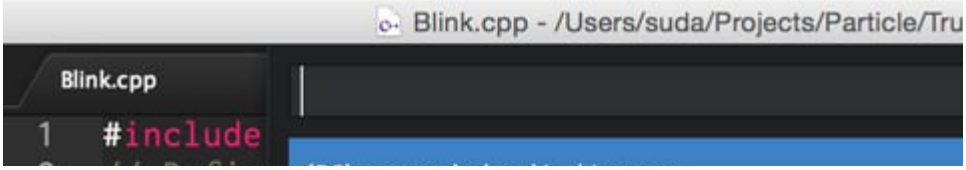
Notes:

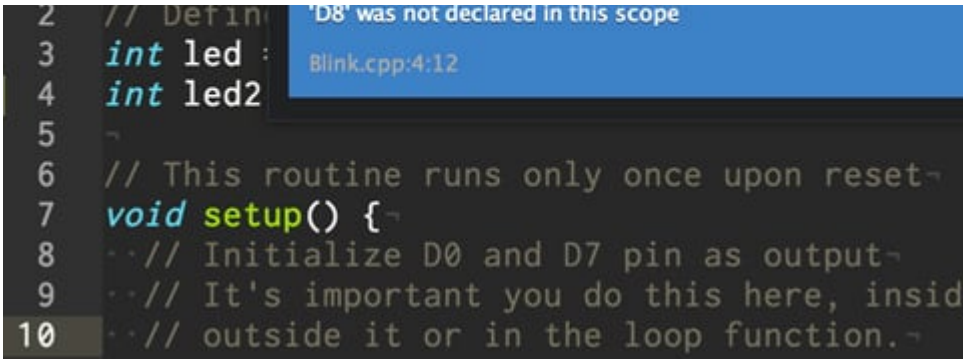
- If other files not related to your project are present in the project directory, you may experience errors when trying to compile.
- All the files have to be on the same level (no subdirectories) like this

To compile your current project, click on the **Compile in the cloud** button. If your code doesn't contain errors, you'll see a new file named **PLATFORM_firmware_X.bin** in your project's directory (where *PLATFORM* is name of currently selected platform and *X* is a timestamp).

Different devices usually require separate binaries (i.e. you can't flash Core with firmware compiled for a Photon) and resulting file is going to be compiled for platform of currently selected device. If you don't have a device selected, the code is going to be compiled for the Core.

NOTE: Remember that ***.cpp** and ***.ino** files behave differently. You can read more about it on our support page.





If there are some errors, you'll see a list of them allowing you to quickly jump to relevant line in code. You can show this list by clicking red error icon on the bottom of the window.

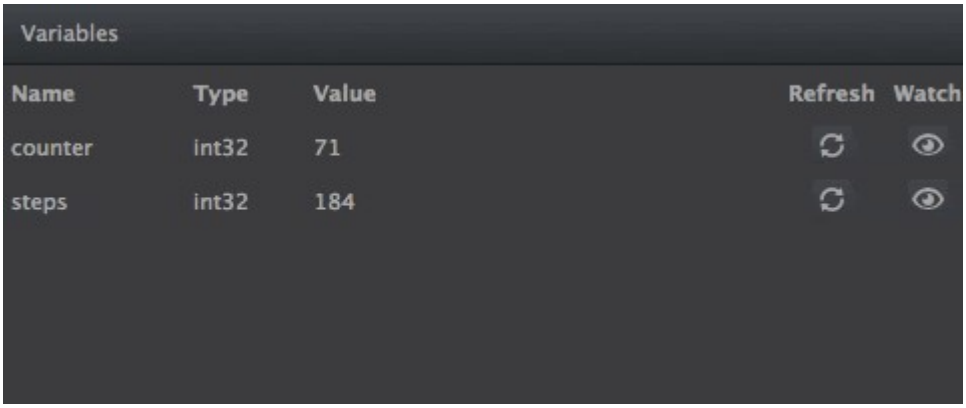
Flashing device

When you're sure that your code is correct it's time to flash it to the device. To do this, click **Flash using cloud** button. Your code will be sent wirelessly to your device. If the request was successful, the LED on your device will begin flashing magenta as code is downloaded to it. The process is complete when the magenta is replaced by your online status indication patterns.

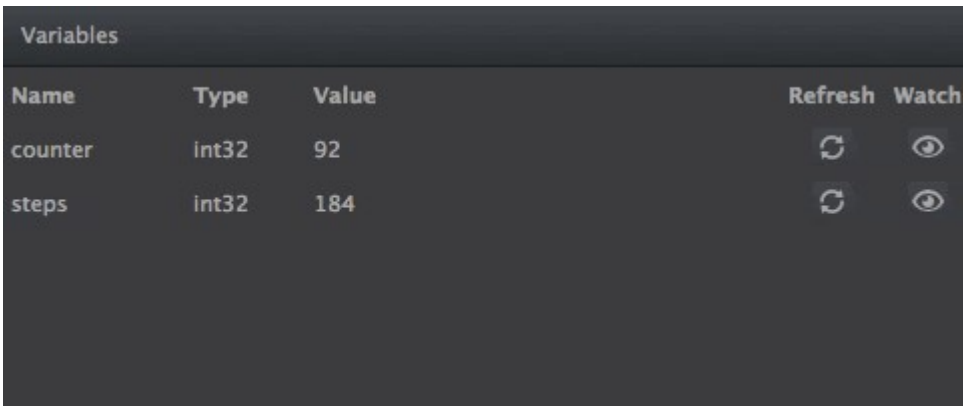
Cloud variables & functions

To access all registered variables and functions, go to **Particle** -> **Show cloud functions/Show cloud functions** menus.

Variables



All variables declared with Particle.variable() are shown on the left side of panel. To poll latest variable value, click **Refresh** button for variable you want to update.



When you want to check variable value constantly, you can click **Watch** button. When a variable is watched, Particle Dev will fetch latest value every 5 seconds.

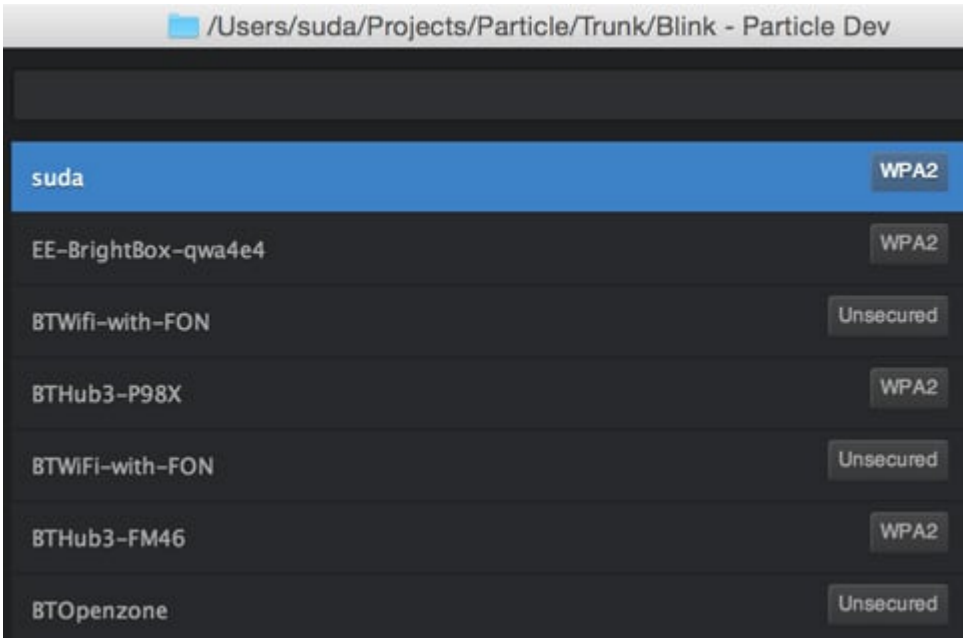
Functions

To call an exposed function simply click on the button with its name. You'll see any data the function returns on the right side.

You can also add parameters to the call by entering them to the right of button.

Managing Your Device

Setting up Wi-Fi

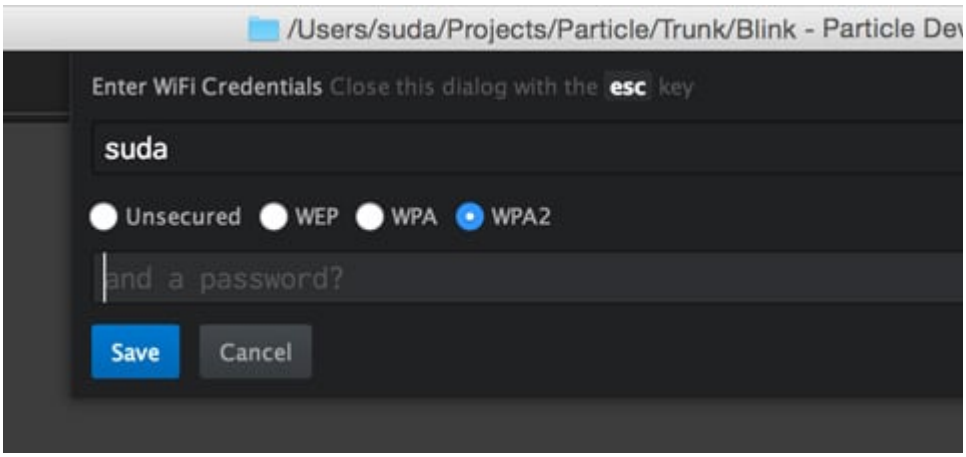


To setup device's Wi-Fi, connect it via USB and click **Setup device's Wi-Fi...** button on the toolbar.

If your device isn't in you'll see animation showing how to enter that state.

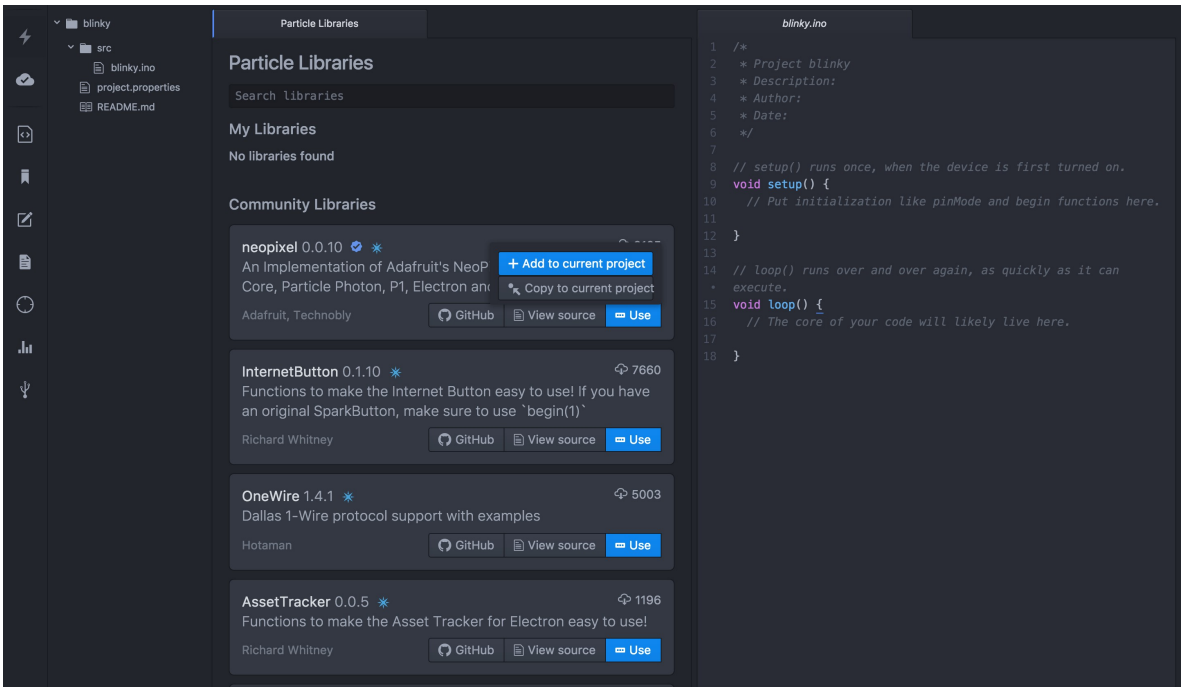
Next you'll see all available networks. The one you are currently connected to will be listed first.

Select the one you want your device to use or choose **Enter SSID manually** (listed last) to specify all information by hand.



Now you need to fill missing information and click **Save**. Your device will go dark for a second and then try to connect to the Wi-Fi.

Using Community Libraries



Firmware libraries are an important part of how you connect your Photon or Electron to sensors and actuators. They make it easy to reuse code across multiple Particle projects, or to leverage code written by other people in the Particle community. As an example, firmware libraries make it easy to get data out of your DS18B20 temperature sensor without writing any of the code yourself.

Particle libraries are hosted on GitHub, and can be easily accessed through through all of Particle's development tools including the Web IDE.

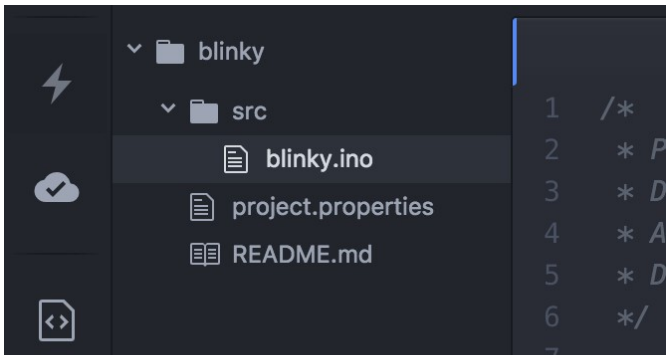
To include a firmware library in your Particle project, open the library drawer in the Desktop IDE, search for the corresponding library for your sensor or actuator, click the `Use` button, then select `Add to current project`. Adding a library in your project will add the library dependency to the `project.properties` file that will be compiled with your project when it is verified or flashed to your target device.

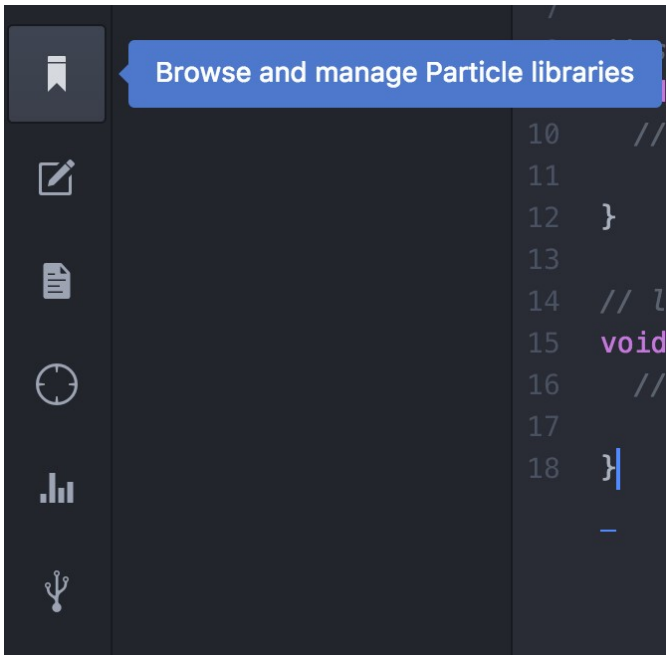
Read on for detailed instructions to include a firmware library in your Particle application with Build.

We have a detailed reference guide about libraries but for now here's a step by step guide on how to include a library in our Desktop IDE.

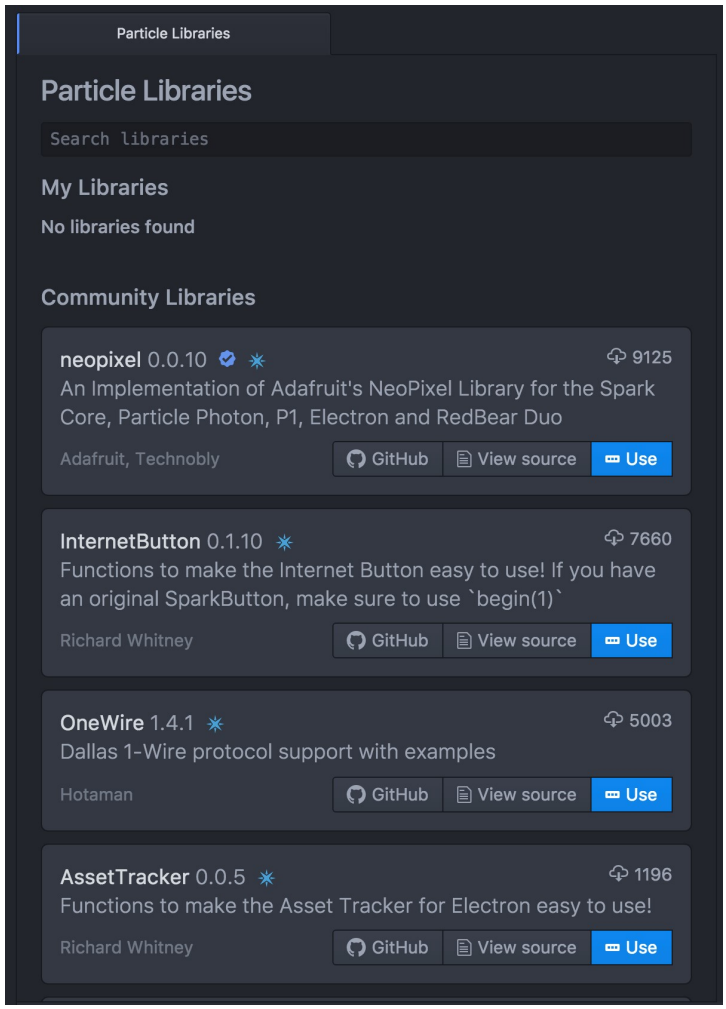
Step 1 - Open the libraries tab

Once you have opened your Particle project in the Desktop IDE, open the libraries tab by clicking on the `Browse and manage libraries` button on the left hand toolbar.





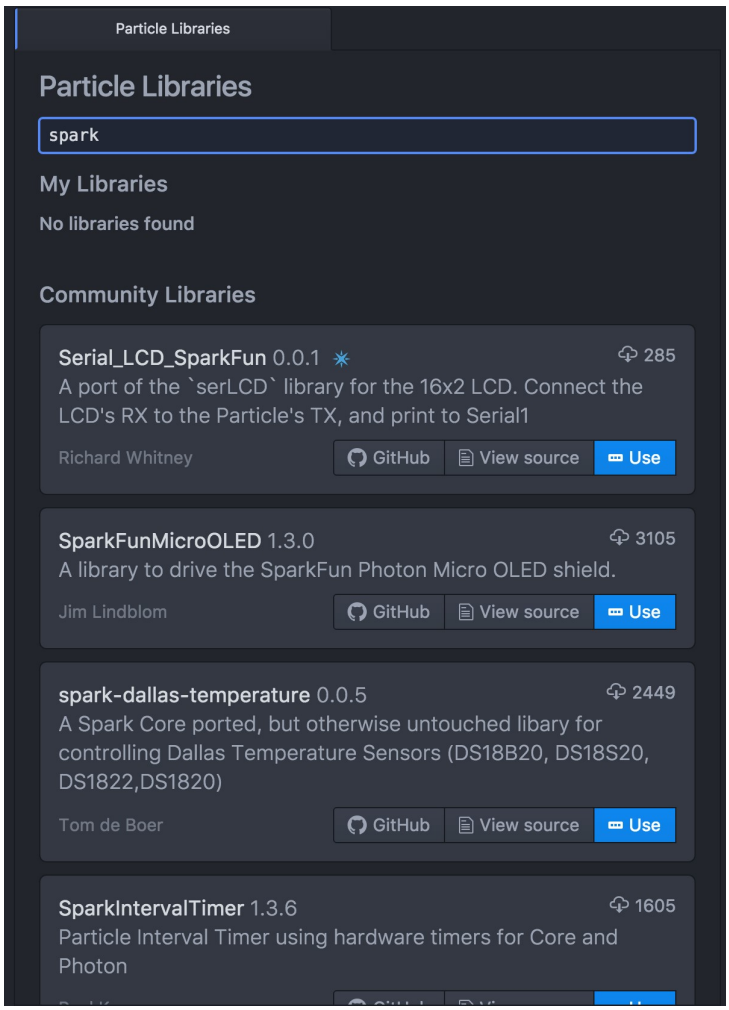
Step 2 - Find the library you need



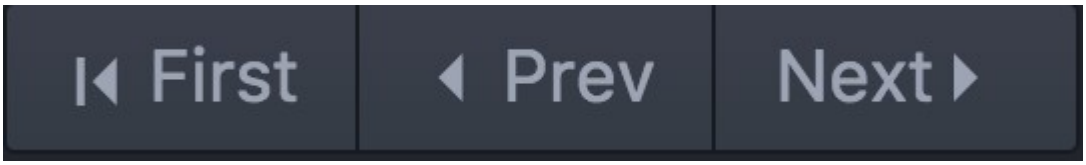
Once you open the libraries tab, you'll be presented with a list of libraries. Libraries with the Particle logo next to them are Official libraries created by the Particle team for Particle hardware. Libraries that have a check mark next to them are Verified libraries. Verified libraries are popular community libraries that have been validated by the Particle team to ensure that they work and are well documented. Click here To learn more about the different kinds of Particle libraries.

To find the right library for your project, you can either search for it directly or browse through popular firmware libraries using the browsing buttons at the bottom of the library list.

Search. To search for a library, begin typing in the search bar. Search results are ranked by match with the search term with a preference for official and verified libraries.

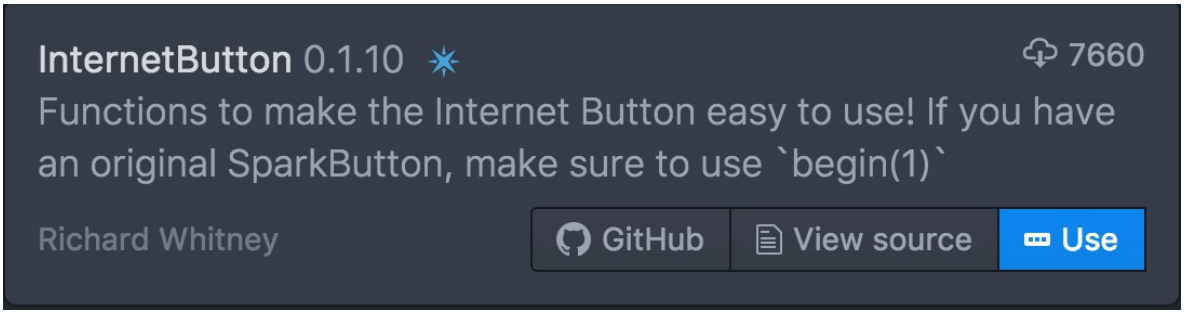


Browsing buttons. Not sure what library you're looking for? Use the browsing arrows beneath the library list to view additional Particle libraries in our firmware library manager. Pagination also works with search results.



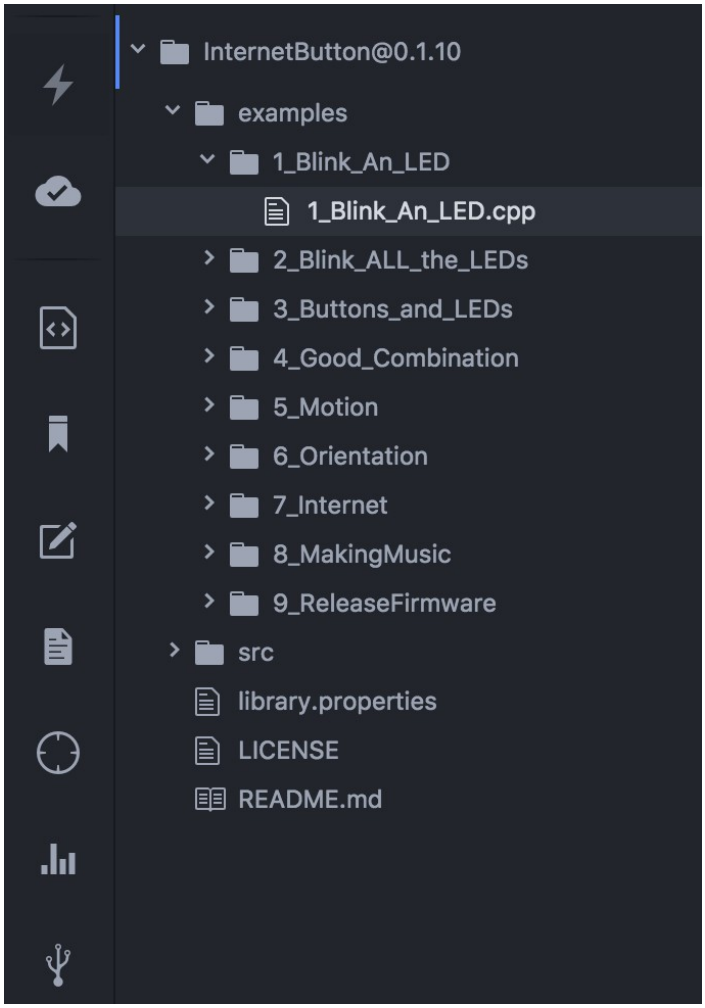
Step 3 - Library details

All the information you need to select your library is available in the search result cards for each library.



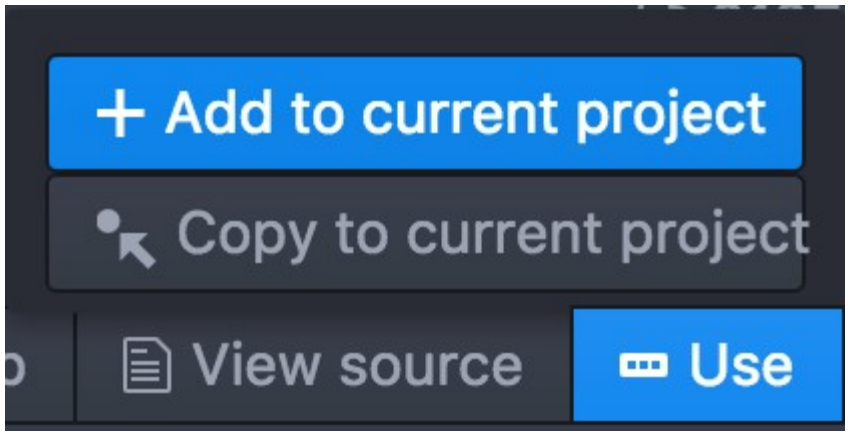
The information included with each library search result includes:

- **`Library name`** : The name of the library. The name must be unique, so there aren't two libraries with the same name.
- **`Library version`** : The version of the library. This follows the semver convention.
- **`GitHub link`** : Where the library is hosted. The code for public libraries must be open-sourced. See how to Contribute a library.
- **`Library description`** : Detailed information about the library
- **`View source`** : Clicking this icon will download the source files of the library and open them in another window. Library source files include the source files for the library itself which follow the new library file structure, as well as library examples, which demonstrate usage of the library.



- `Install count` : This is the number of times a particular library has been added to a Particle project

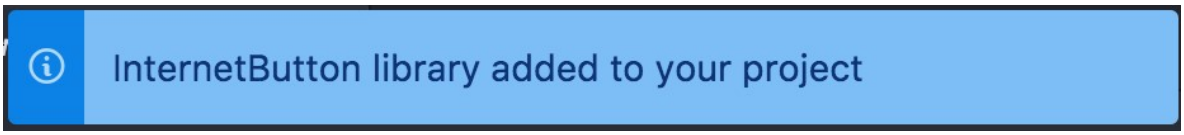
Step 4 - Click on `Add to current project`



To add a firmware library to a project, click the `Use` button. You will be presented with two options -- `Add to current project` or `Copy to current project`.

- **Add to current project** will include the library as a line in your project's project.properties file and will be included by the Particle compiler when your project is verified or flashed.
- **Copy to current project** will download a local copy of the source files of the library to your project's `src` folder. The library can be inspected and modified before it is sent to the Particle compiler. If you copy a library into a project, the library files must be included in the `src` folder or they will not be compiled with the rest of your project.

Once you add the library to your Particle project, you should see a confirmation message



the library name and version number should be added to the `project.properties` file for your Particle project.

To make the library functionality available to your application, you add an include statement to your application source code. The include statement names the library header file, which is the library name with a `.h` ending.

For example, if we were using the library "UberSensor", it would be included like this:

```
#include "UberSensor.h"
```

Congrats! You have now added a firmware library to your Particle project in the Desktop IDE!

Contribute a library

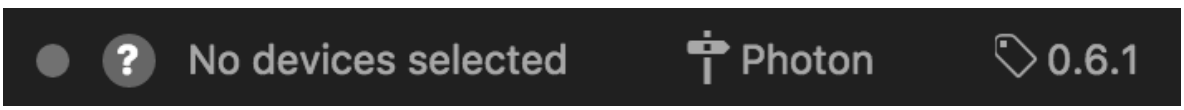
See the detailed library guide to find more about contributing a library from the Desktop IDE.

Targeting different platforms and firmware versions

Similarly to the Web IDE you can specify exactly which platform (Core, Photon, Electron or others) you're using and at which specific firmware version your project is depending.

Note: By default all projects are compiled for latest version of firmware for a Photon.

To know what platform and version you are targeting take a look at the status bar:



The first item is currently selected device. Once you select a different device, the target platform will be automatically changed to its platform.

The second one is the platform you want to target. Different platforms have different capabilities (i.e. Photon has WiFi but Electron has cellular instead) so

keep in mind that some firmware methods might not exist or work differently (consult the reference to make sure they will work as you expect).

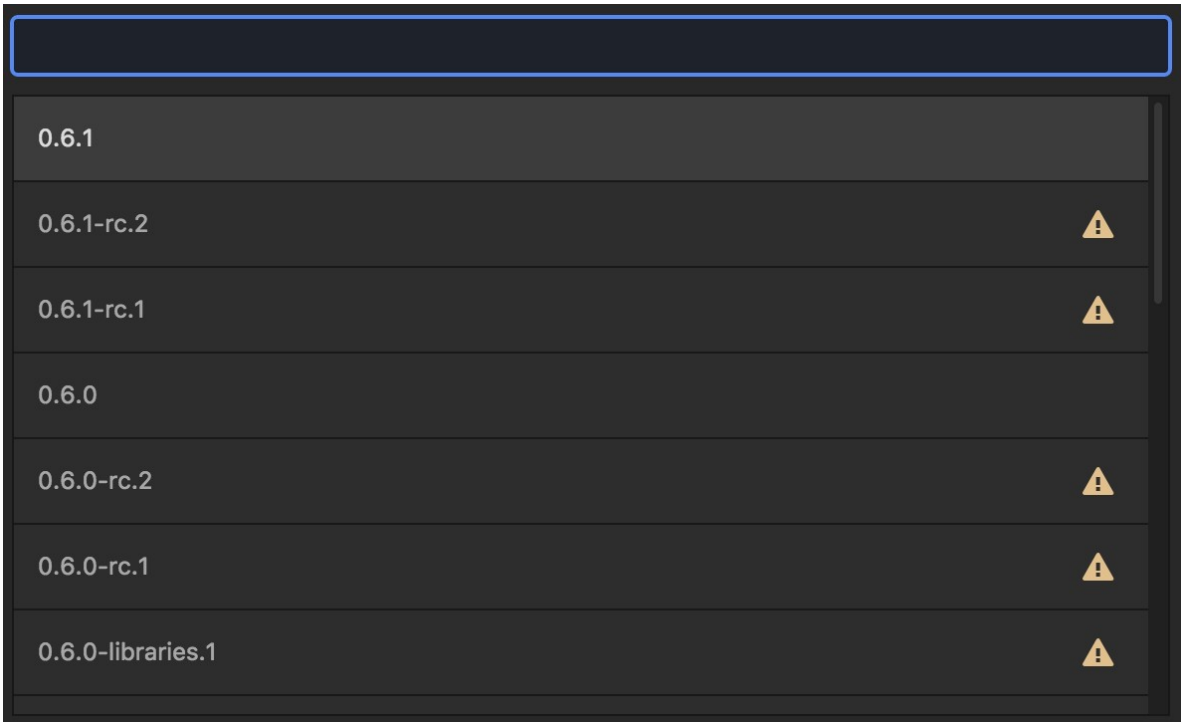
Clicking on the platform name will allow you to select a different one:

Note: You can target platforms you don't own hardware of and at least test if the code compiles.

The last item is the firmware version aka the build target. This allows you to select exactly which version you want to use. When reading the reference you might have noticed that some functions have a note saying: *Since X.Y.Z*. This specifies the minimum build target you need to use in order to have this function available.

When you use a version newer than's on your device (which can be checked using `particle serial inspect` CLI command) it will enter **safe mode** which should be automatically fixed with multiple consecutive flashes. The exception here is the Electron where updating system firmware would incur charges. In this case, the IDE will select the build target that's currently on the device in order to keep the device running.

Clicking on the build target will show the list of available ones for current platform:



You can see that some of the build targets have a warning sign next to them. Those are pre-releases. They will bring the latest features but might not be as stable as the other releases. We encourage you to test them but only for experimental purposes (i.e. not in production).

CONSOLE

The Particle Console is your centralized IoT command center. It provides interfaces to make interacting with and managing Particle devices easy. This guide is divided into two main sections, tools for developers and tools to manage product fleets.

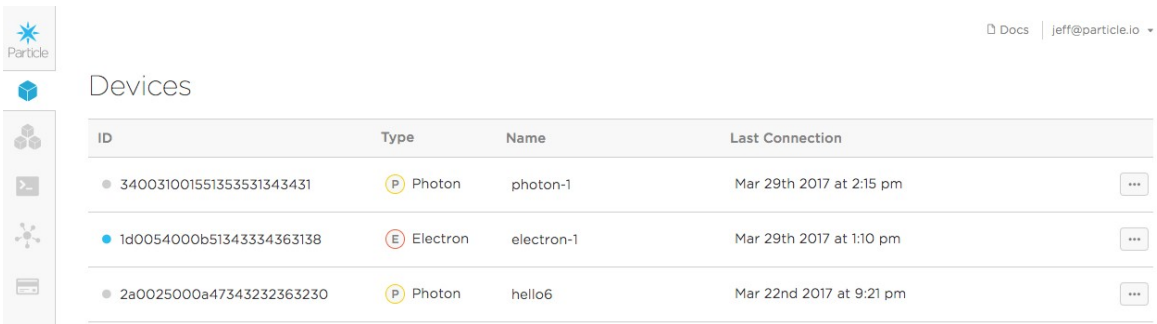
Note: The Console does not yet work in Microsoft Internet Explorer including Edge. Please use another browser, such as Chrome or Firefox, to access the Console. If you're experiencing rendering issues, turn off any ad blocking extensions you may be using.

Developer Tools

While actively developing an IoT project or product, the Console offers many helpful features to make prototyping a breeze. See the last time a device connected, debug a firmware issue by observing event logs, set up a webhook to send data to an external service, and more.

Devices

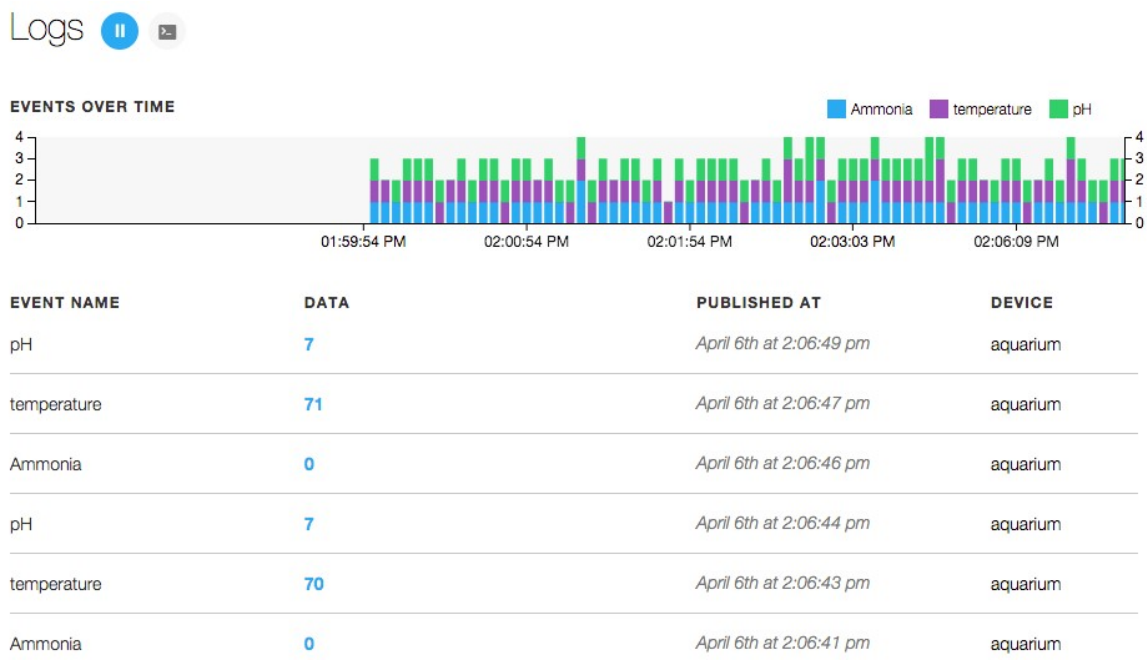
The Devices page allows you to see a list of the devices associated with your account. Here, you can see specific information about each device, including it's unique Device ID, it's name, the type of device (i.e. Photon or Electron) the last time it connected to the Particle cloud, and whether or not the device is currently online.



You can also take certain actions on devices from this view, such as renaming the device and unclaiming it from your account.

Logs

You can watch events published from your devices with `Particle.publish()` come in, in realtime.



The Logs feature provides a clean interface to view event information in real-time, just from your devices. We're hoping that this is handy both while debugging code during development, and checking out recent activity on your device once you power-on your finished project. Here's a snapshot of a Particle device monitoring the health of a theoretical aquarium.

Let's look at it starting at the top.

Near the title, we've got a button to pause and un-pause the event stream.

There's also an icon of a terminal window. When you click on it, we give you a hint on how to get the same information from the API.

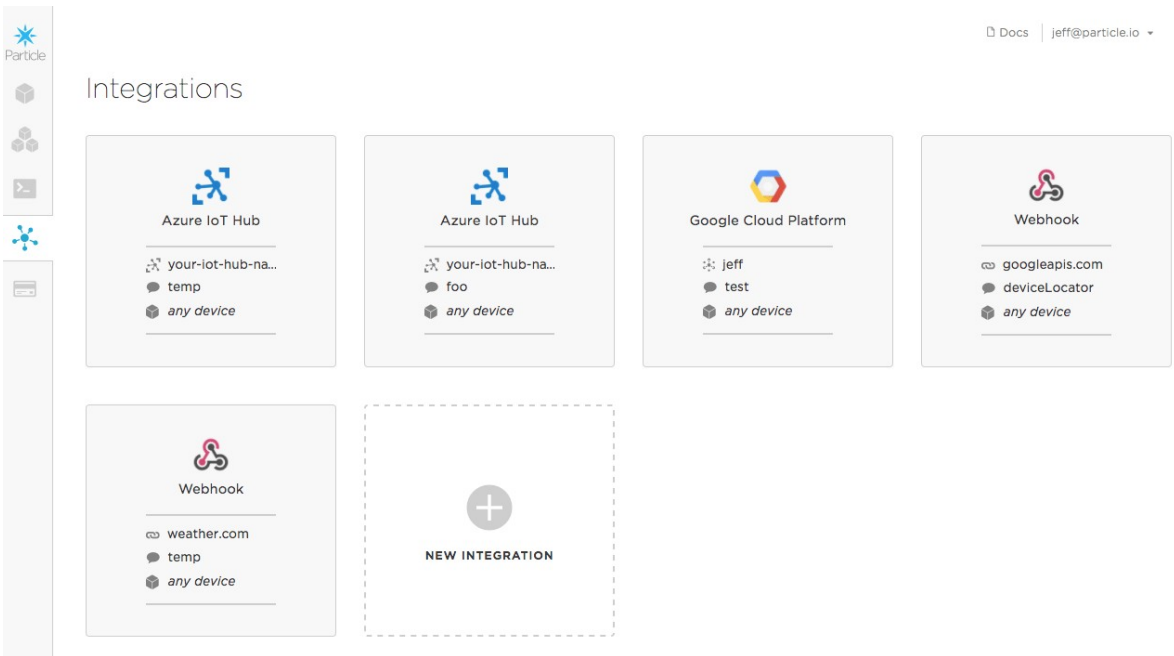
Below the title is a side-scrolling bar graph. It's a visualization of the number of events from your devices over time. Each color in the bar graph represents a unique event name. Each bar is 5 seconds in duration.

At the bottom is a real-time log of events passing through the cloud. You'll get the name, data, timestamp and the device name associated with each event as it comes in. Oh Yeah! And, if you click on the event, you can see a raw JSON view of the event.

In this view, you'll only see events that come in while the browser window is open.

Integrations

Integrations allow you to send data from your Particle devices to external tools and services. The Console provides an interface to create, view, edit, and delete Particle integrations.



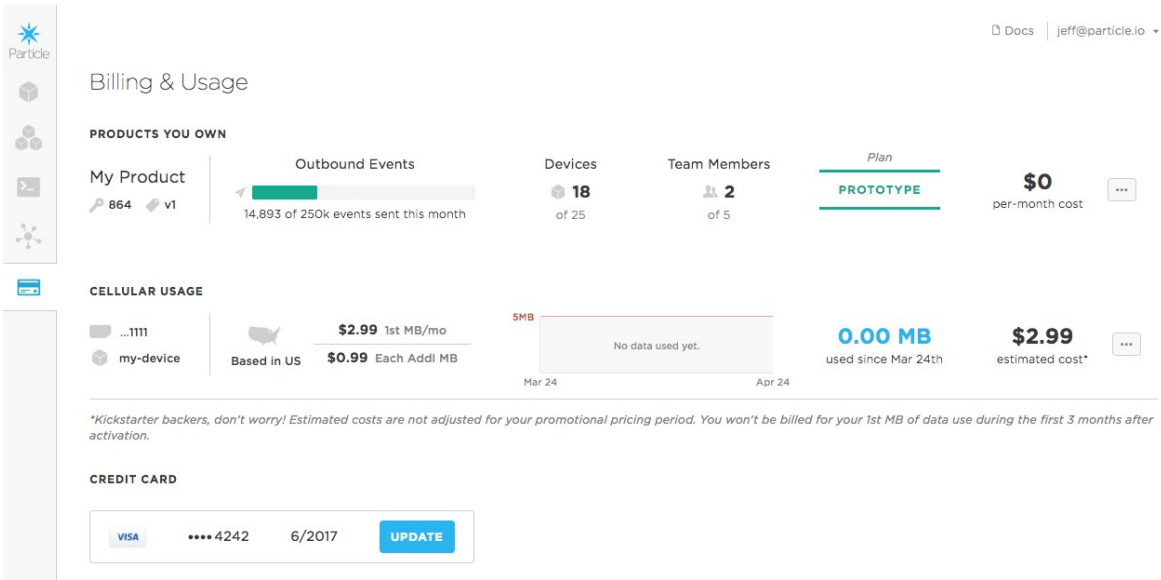
For more information on how to start using integrations, you should check out:

- Webhooks guide
- Webhooks tutorial
- Azure IoT Hub tutorial
- Google Cloud Platform tutorial

Billing

You should also use the Console to view and manage billing information. Some of the things to use the billing view for include:

- Updating the credit card used to pay for Particle platform features
- Viewing SIM card data usage, and managing the connectivity state of the SIM card
- View product billing and usage information, including outbound event count, device fleet size, and current plan



Product Tools

For many using Particle, the end-goal is to move from a single prototype to a professional deployment of thousands or millions of units in the field. When you begin making this transition to managing a larger fleet of devices, you'll find yourself asking questions like:

- *How many* of my devices are online right now?
- *Which* firmware version is running on each device?
- *Who* of my customers are using their devices, and who isn't?
- *Who* in my company has access to this fleet, and what information can they access?

This is where creating a Particle product is vital to ensure scaling can happen seamlessly and successfully.

Luckily, the Particle Console is designed to give you full visibility into the state of your product fleet, and provide a centralized control panel to change how devices are functioning. It allows you and a team to manage firmware running on your devices, collect and analyze product data, and manage team permissions from a single administrative interface.

The first step to get started is understanding the differences between your personal devices and those added to a Product.

Devices vs Product Devices

Up until now, you've been an individual user of Particle. Your devices belong to you, and you can only act upon one device at a time.

When you create a **Product**, you'll have a few additional important concepts available to you: **devices**, **team members** and **customers**.

First, you'll set up a **Product**, the overarching group responsible for the development of your Internet of Things products.

Defining a Product is what unifies a group of homogeneous devices together, and your Product can be configured to function exactly how you envision.

Each Product has its own fleet of associated **devices**. Any hardware on the Particle Cloud including the PØ, P1, Photon, and Electron, could be used inside a Product, but it's important to note that only one type of device will be in each Product

Customers own a device, and have permissions to control their device. You will define the extent of their access to the device when you configure your Product.

Your Product also has **team members** with access to the Console.

It is important to note that *team members* and *customers* have different levels of access. For instance, only *team members* will typically be able to send an over-the-air firmware update, while *customers* may have the ability to control their own product. These access levels will be controlled through the Console.

Defining a product

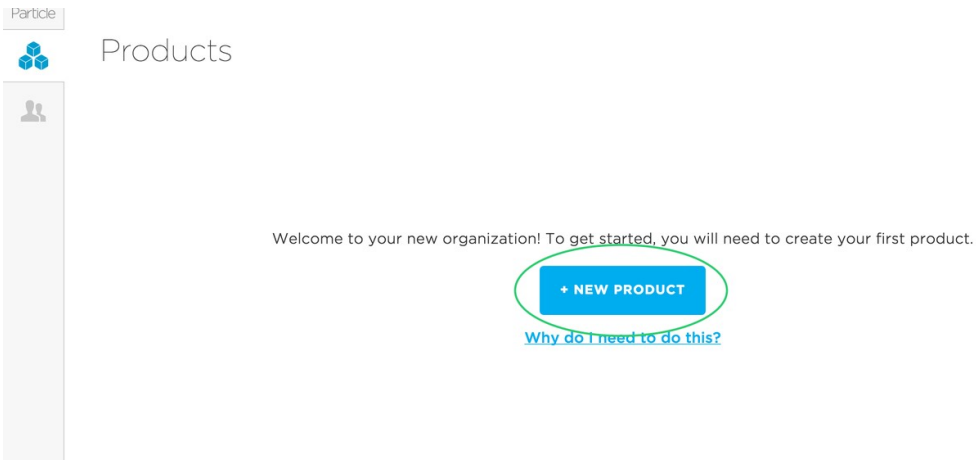
Our cloud platform thinks that all devices are *Photons*, *Electrons*, or *Cores* – unless it's told otherwise. Now's the time to define your own product within the platform and tell us a bit about how that product should behave.

Photons are development kits. They're designed to be easy to reprogram and run a variety of software applications that you, our users, develop.

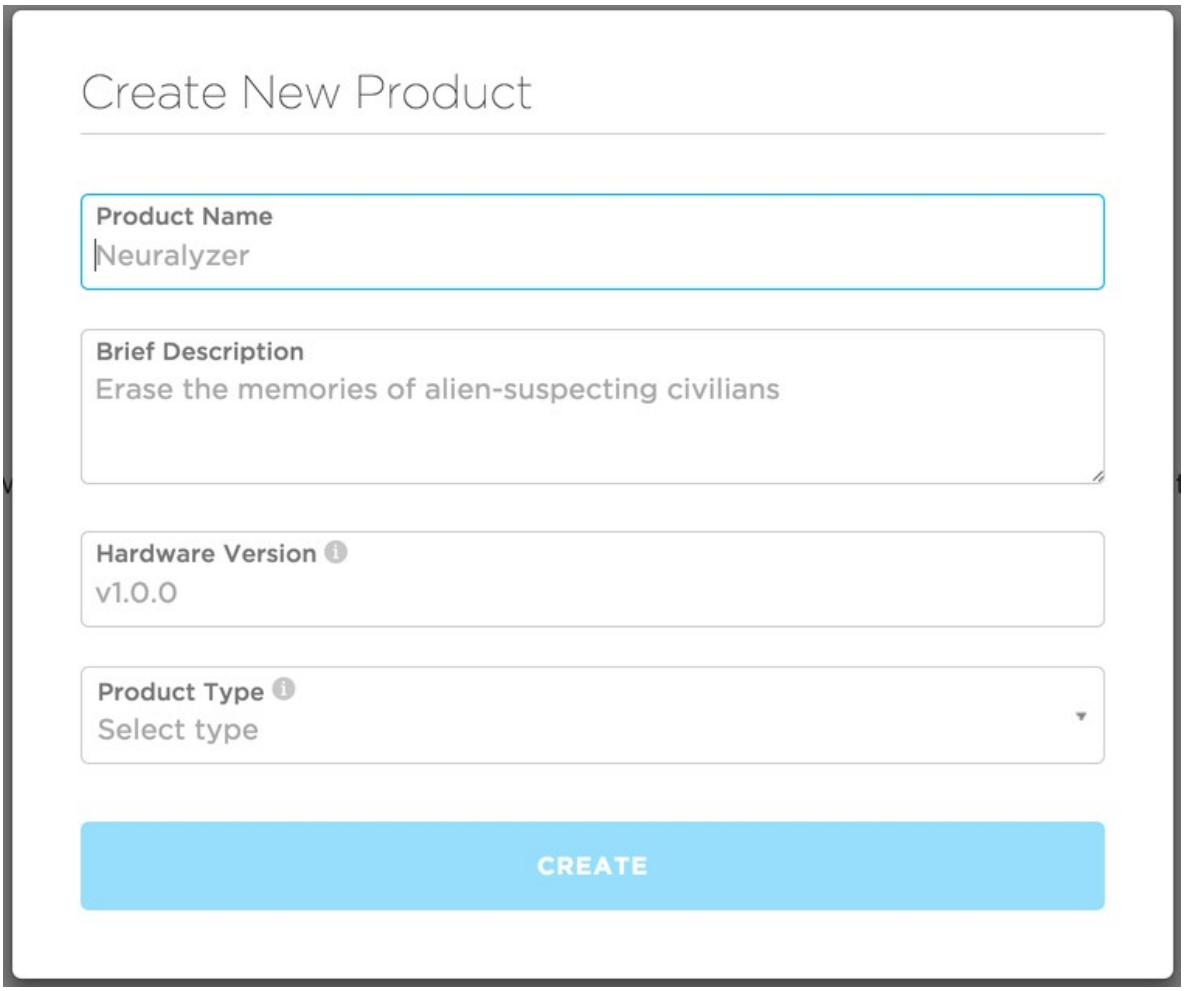
Your product is (probably) not a development kit. While some of the characteristics of the development kits will carry over, you're going to want to make a bunch of changes to how your product works. These include:

- Limiting access (e.g. only certain people can reprogram them)
- Collecting bulk data, events, errors, and logs from all of your devices
- Distributing firmware updates in a controlled fashion

To create a product, return to your personal console page and click on the **New Product** button.



This will open up a modal where you can add basic details about your product:

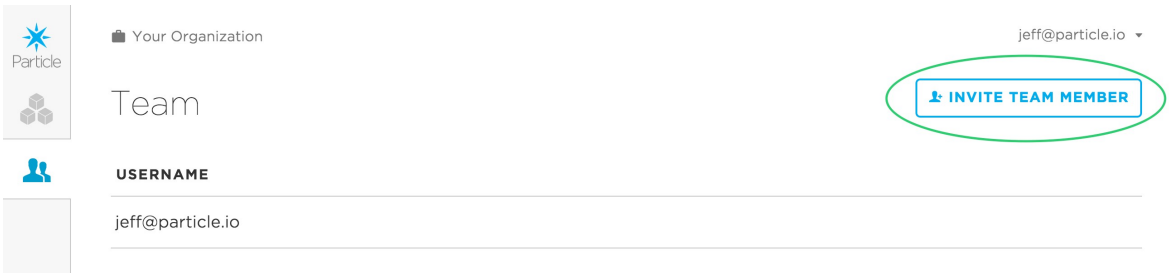


You now have your very first Particle product! Woot!

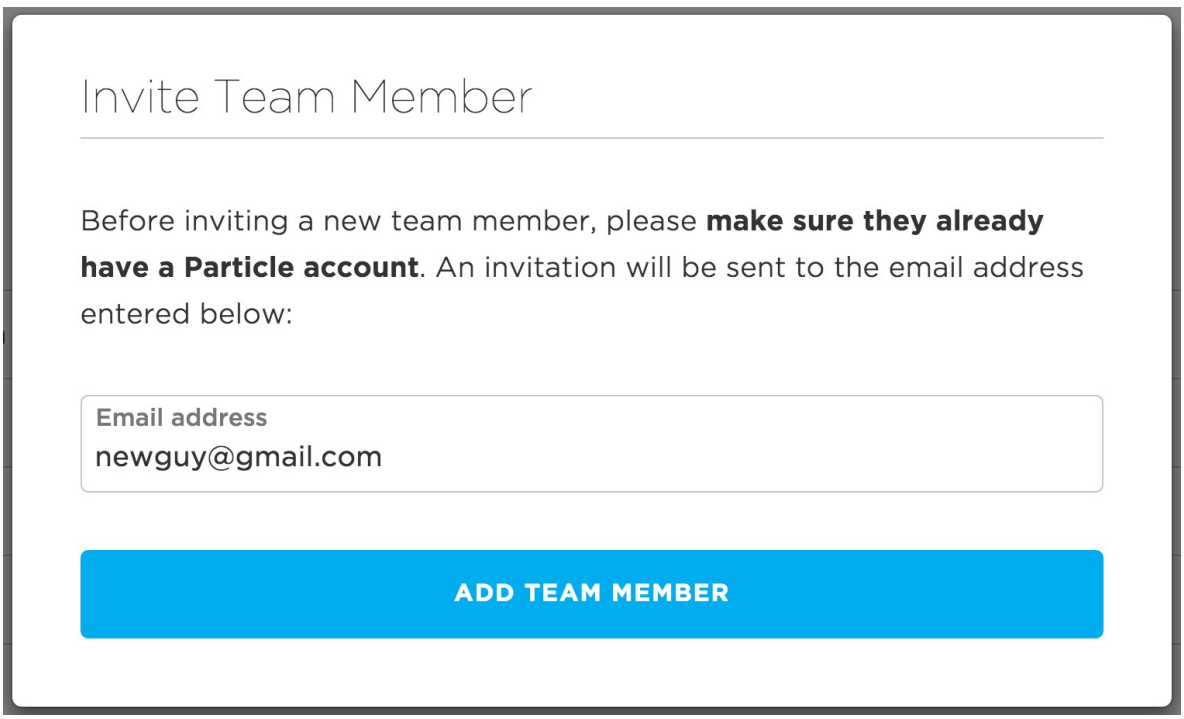
Adding team members

Now that you have created a Product successfully, it's time to add your coworkers and friends that are collaborating with you on your IoT product. Adding a team member will give them full access to your Product's Console.

To do this, click on the *team icon* (👥) on the sidebar of your Product Console. This will direct you to the team page, where you can view and manage team members. Right now, your username should be the only one listed as a member of the Product. To add a new team member, just click the *Invite team member* button pictured below:



Clicking this button will open up a modal where you can invite a team member by email address. Before inviting a new team member, **make sure that they already have a Particle account with the email address you will be using to invite them to the Product**.



The invite team member modal

Once your team member is successfully invited, they will receive an email notifying them of their invitation. The next time they log into their Console, they will have the option of accepting or declining the invitation. Remember that you can have up to 5 team members in the free Prototype tier, so go send some invites!

Nice! Now you have a Product with a team.

Your Product ID

When you created your product, a unique numeric ID was assigned to it. This small piece of information is *very, very important* to you as a product creator, and it will be used countless times during the development and manufacturing process for your product. You will be able to find your product's ID at any time in the navigation bar when viewing information about your product:

Your product ID is marked with a key icon

This ID will be used by the Particle Cloud to identify which devices belong to your Product, and subsequently it is part of what empowers you to manage firmware running on those devices *en masse*.

When working with devices that belong to your Product, it is important to note that this product ID must be compiled into the firmware that is running on each device. The product ID that the device reports to the cloud from its firmware will determine which Product it requests to become a part of. This will be covered more in-depth in the rollout firmware section below.

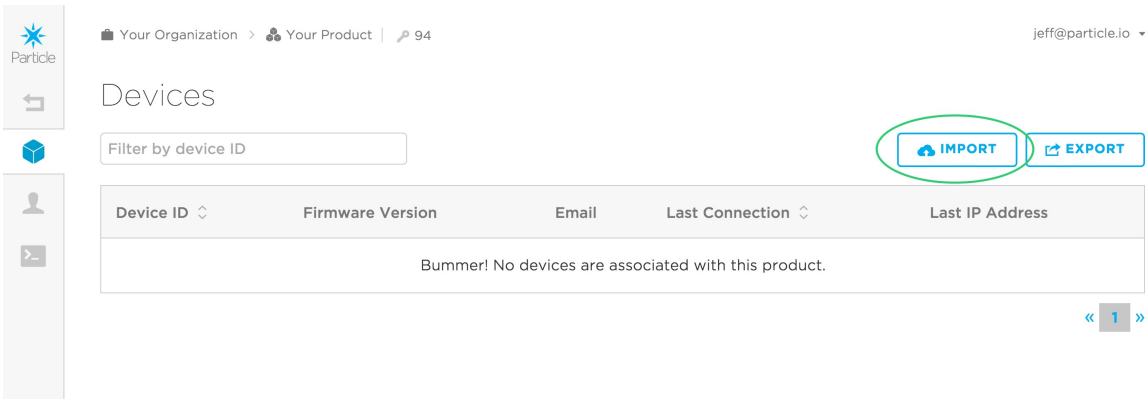
Adding Devices

Now that you have your Product, it's time to import devices. Importing devices will assign them to your Product and allow you to start viewing and managing these devices within your Product Console.

For any product you may be developing, you likely have one or more Particle development kits (i.e. a Photon) that you have been using internally for prototyping purposes. We strongly recommend importing these devices into your Product, and using them as your *development group*.

In addition, you'll want to have a *test group* of devices to serve as the guinea pigs for new versions of product firmware. You should get into the habit of uploading a new version of firmware to your product, and flashing it to your test group to ensure your code is working as expected. This too will be covered more in-depth in the rollout firmware section below.

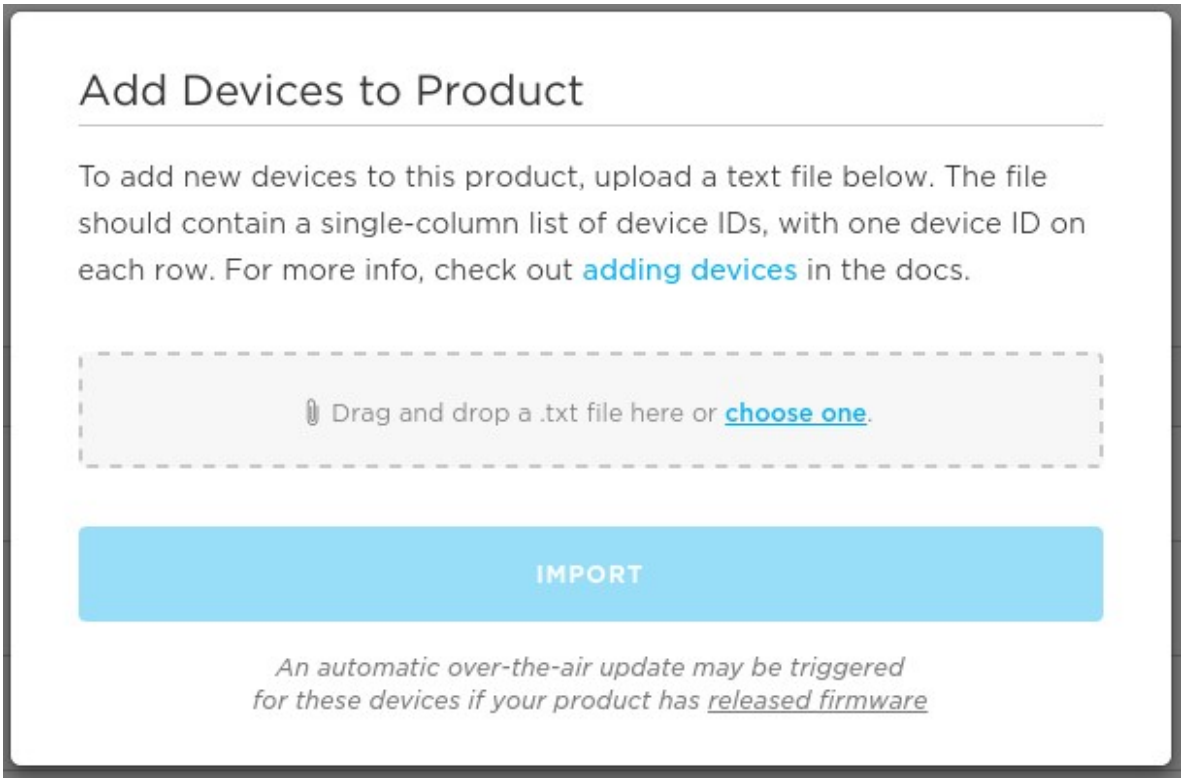
To import devices, click on the Devices icon in your product sidebar, then click on the "Import" button.



To allow you to import devices in bulk, we allow you to upload a file containing multiple device IDs. Create a `.txt` file that contains all of the IDs of devices that you would like to import into your product, one on each line. Not sure what your device ID is? *You cannot register devices that have already been 'claimed' by someone outside of your team; all of these devices must either belong to a team member or belong to no one*. The file should look something like this:

```
55ff6d04498b49XXXXXXXXXX
45f96d06492949XXXXXXXXXX
35ee6d064989a9XXXXXXXXXX
```

Where each line is one Device ID. Once you have your file ready, drop it onto the file selector in the import devices dialog box.



As noted at the bottom of the dialog box, if you previously rolled out firmware, those newly imported devices will be updated over the air to that firmware next time they connect to the Particle Cloud.

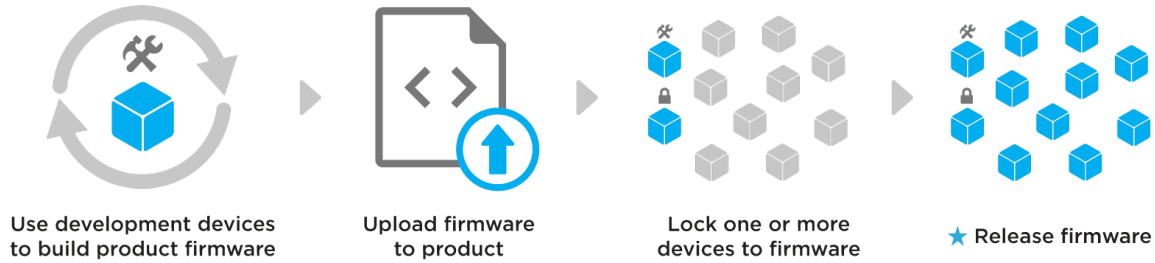
Rollout Firmware

One of the most valuable features of a Particle product is being able to seamlessly manage firmware on a fleet of IoT devices. You now have the ability to continuously improve how a device functions after deployment. In addition, product firmware management allows you to quickly and remotely fix bugs identified in the field, fleet-wide.

This happens through **firmware releases**, which targets some or all of a device fleet to automatically download and run a firmware binary.

Recommended development flow

When releasing firmware your fleet, it's helpful to first understand Particle's recommended release flow. This flow has been designed to minimize risk when deploying new firmware to devices:



The recommended flow for releasing firmware

1. The first step of the release flow is using **development devices** to rapidly develop and iterate on product firmware. These are special product devices marked specifically for internal testing. This gives you the flexibility to experiment with new firmwares while still simulating behaviors of deployed devices in the production fleet. For information on marking a device as a development devices, check out the guide.
2. When you have finalized a firmware that you feel confident in releasing to your fleet, **prepare the binary and upload it to your product**.
3. Before releasing, you will need to ensure that the uploaded product firmware is running on at least one device in your product fleet. Your development device(s) may already be running the firmware, but we also recommend **locking one or more devices** to the newly updated firmware and ensure that it re-connects successfully to the cloud. This is because locking more closely represents a release action, with the specific firmware being delivered to a product device.
4. **Mark the firmware as released**. This will target product devices to automatically download and run the firmware. The Particle Cloud will respect the precedence rules to determine which firmware is delivered to a given

device. If you are on the Enterprise plan with access to device groups, you can more safely roll out the firmware by targeting a subset of the fleet for release.

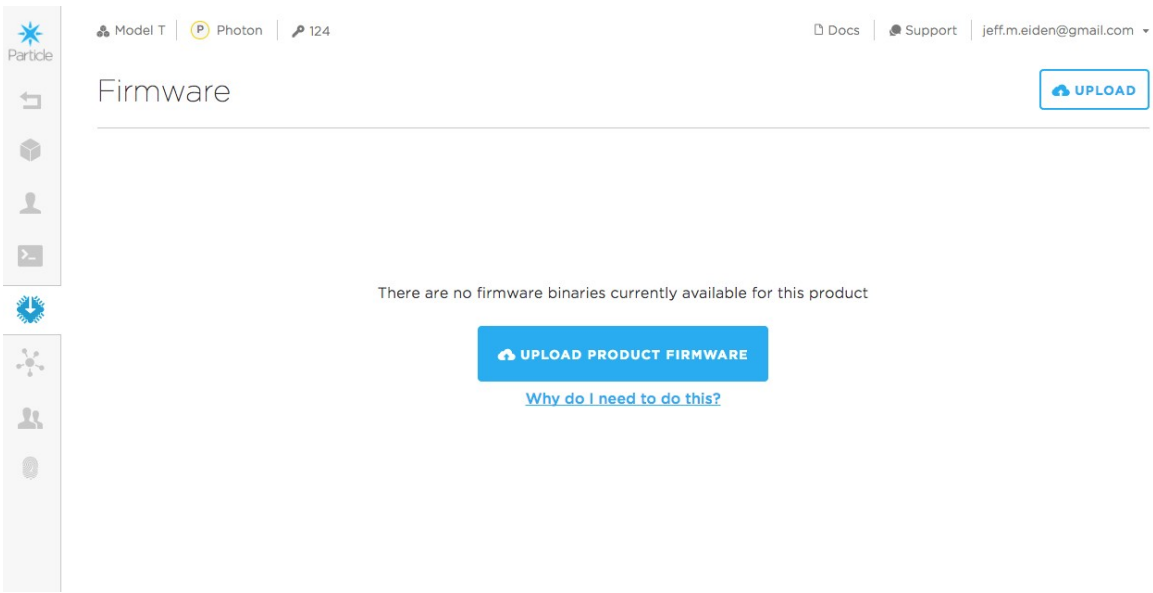
The rest of this section contains details around how to go through this process.

Development devices

Please visit the guide on development devices for information on this feature.

Preparing a binary

Click the Firmware icon in the left sidebar to get started. This will direct you to your product's firmware page, your centralized hub for viewing and managing firmware for your product's devices. If you haven't yet uploaded any firmware for this Product, your page will look like this:



If you have been using the Web IDE to develop firmware, you are used to the process of writing, compiling, and then flashing firmware. You will follow the same high-level process here, but altered slightly to work with a fleet of devices. The first thing you'll need to do is compile a *firmware binary* that you will upload to your Console.

Unlike compiling a binary for a single device, it is critical that the **product ID** and a **firmware version** are included in the compiled binary. Specifically, you must

add `PRODUCT_ID([your product ID])` and `PRODUCT_VERSION([version])` into the application code of your firmware. This is documented fully here.

Add these two "macros" near the top of your main application `.ino` file, below `#include "Particle.h"` if it includes that line. Remember that your product ID can be found in the navigation of your Console. The firmware version must be an integer that increments each time a new binary is uploaded to the Console. This allows the Particle Cloud to determine which devices should be running which firmwares.

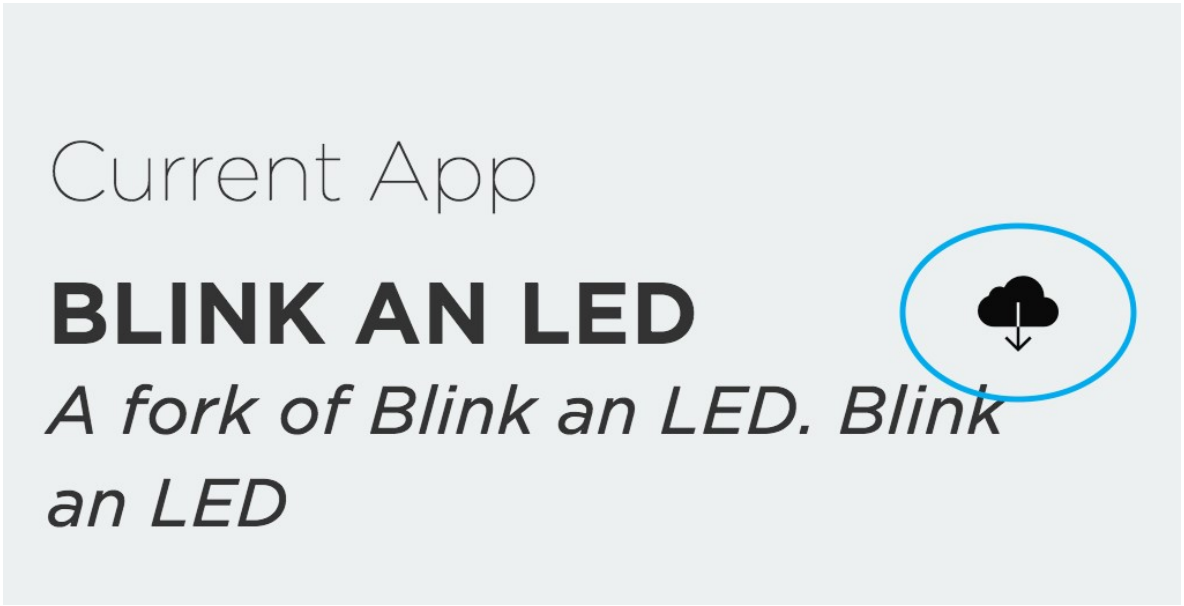
Here is an example of Blinky with the correct product and version details:

```
PRODUCT_ID(94);
PRODUCT_VERSION(1);

int led = D0;  // You'll need to wire an LED to this one to see it
blink.

void setup() {
  pinMode(led, OUTPUT);
}

void loop() {
  digitalWrite(led, HIGH);   // Turn ON the LED pins
  delay(300);               // Wait for 1000mS = 1 second
  digitalWrite(led, LOW);    // Turn OFF the LED pins
  delay(300);               // Wait for 1 second in off mode
}
```

Compiling Binaries

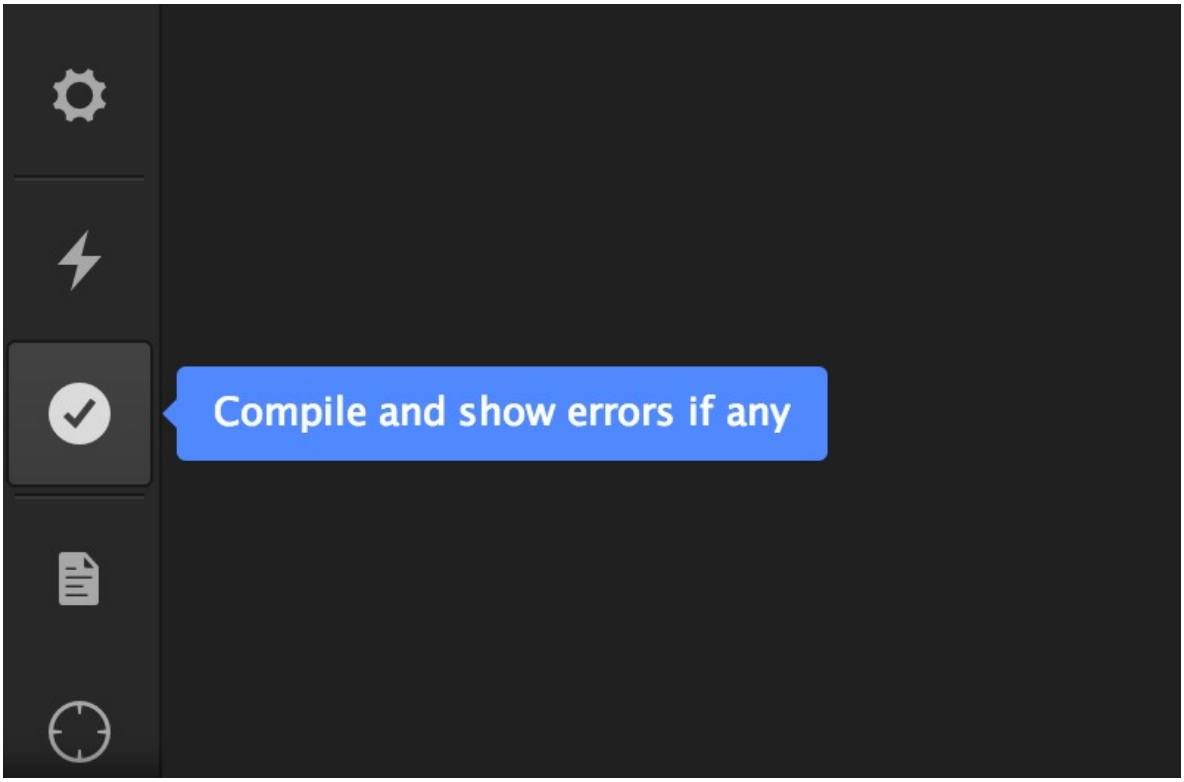
If you are in the Web IDE, you can easily download a compiled binary by clicking the code icon (<>) in your sidebar. You will see the name of your app in the pane, along with a download icon (shown below). Click the download icon to compile and download your current binary.

Particle Apps



Compile and download a product binary from the web IDE

If you are using the Desktop IDE, clicking on the compile icon (✔) will automatically add a `.bin` file to your current working directory if the compilation is a success. **Note**: Make sure that you have a device selected in Particle Dev before compiling.

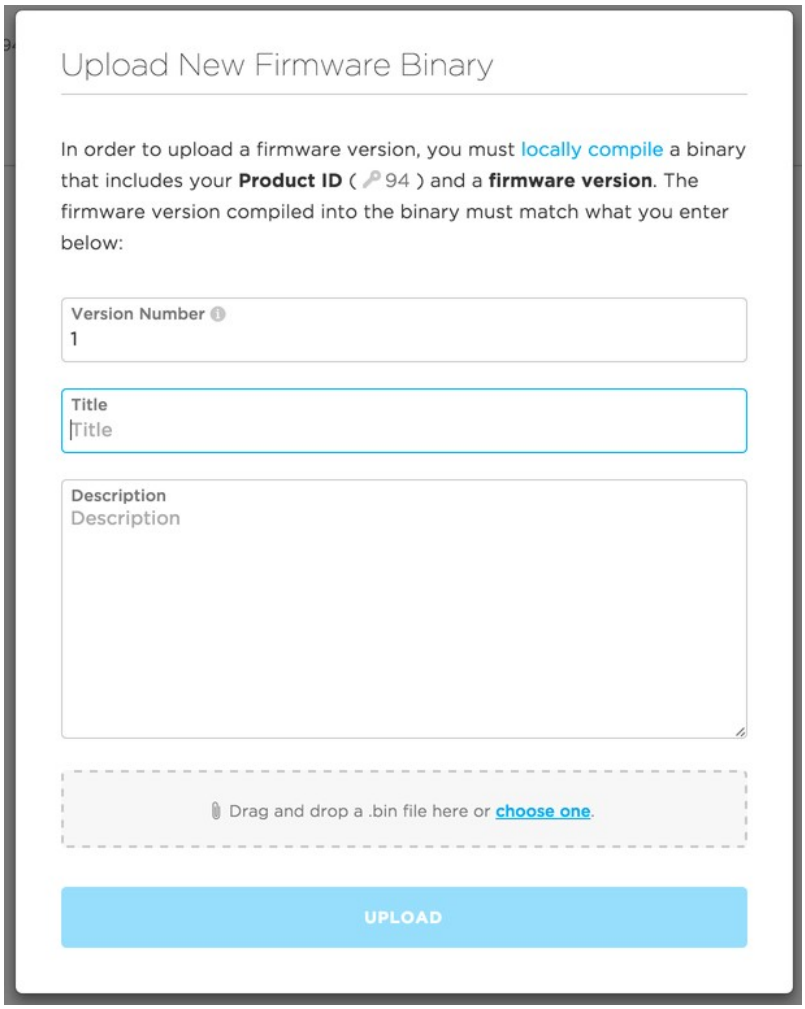


Particle Dev will automatically add a compiled binary to your working directory after you compile

Once you have a binary ready to go, it's time to upload it to the Console!

Uploading firmware

Back on the firmware page, click on the **Upload** button in the top-right corner of the page. This will launch the upload firmware modal:

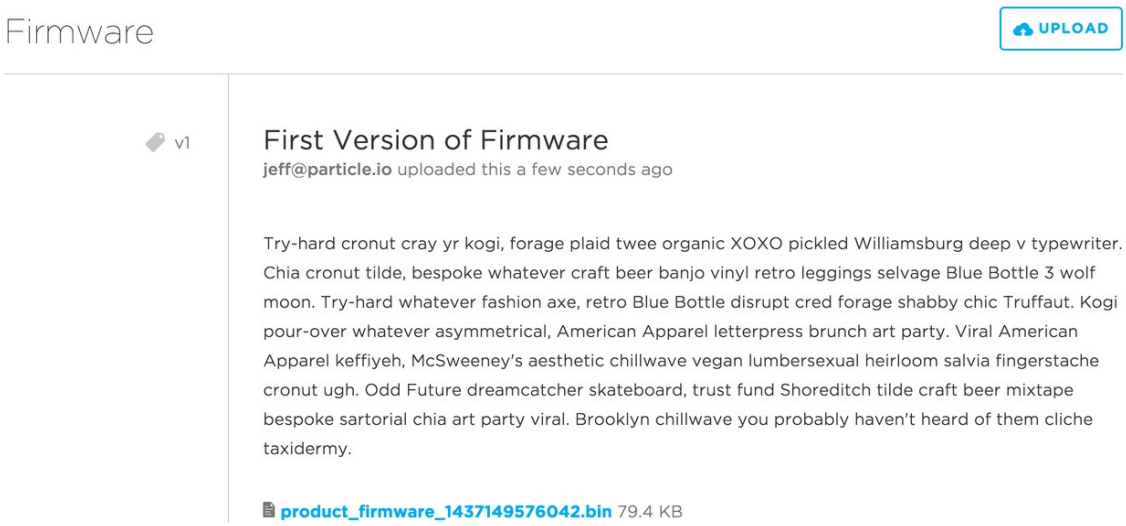


A few things to keep in mind here:

- The firmware version that you enter into this screen **must match** what you just compiled into your binary. Madness will ensue otherwise!

- You should give your firmware a distinct title that concisely describes how it differs from other versions of firmware. This name will be important in how firmware is rolled out to devices
- Attach your newly compiled `.bin` file in the gray box

Click upload. Congrats! You've uploaded your first version of product firmware! You should now see it appear in your list of firmware versions.



Your firmware version now appears in your list of available binaries

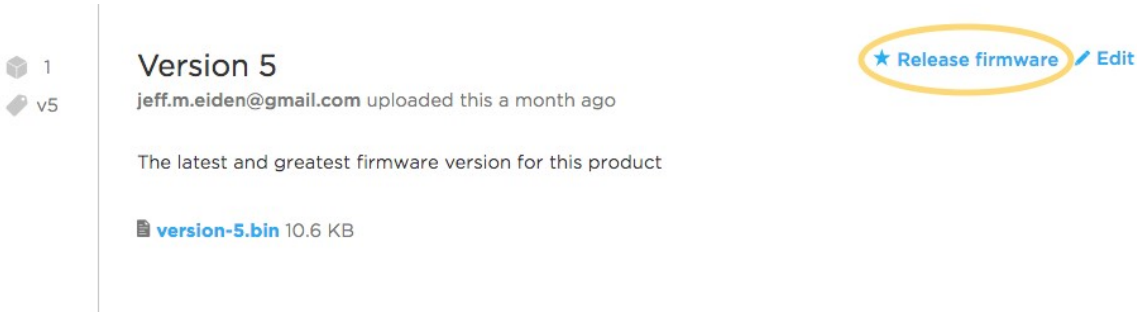
Releasing firmware

Time to flash that shiny new binary to some devices! Notice that when you hover over a version of firmware, you have the ability to **release firmware**. Releasing firmware is the mechanism by which any number of devices can receive a single version of firmware without being individually targeted.

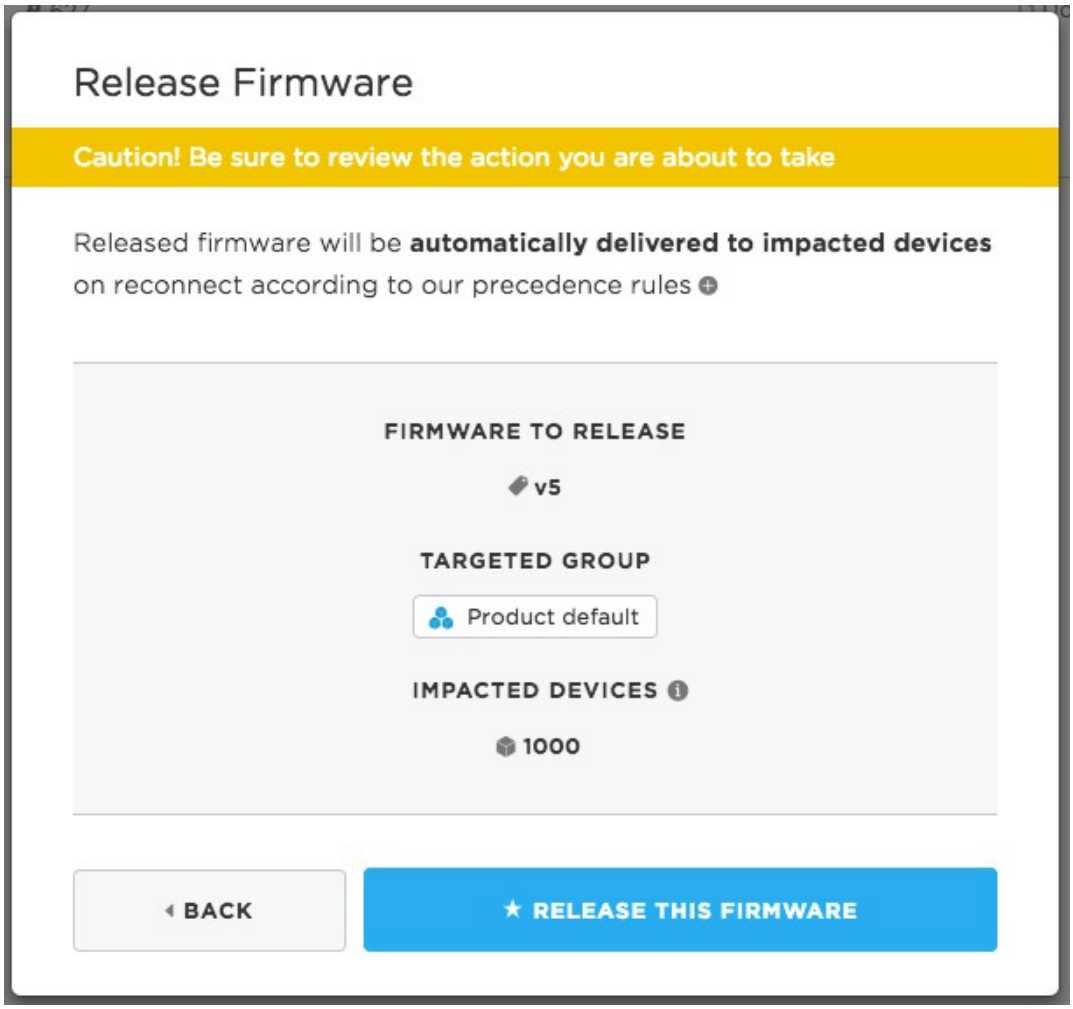
Imagine identifying a bug in your firmware and pushing out a fix to thousands of devices that are out in the field. Or, consider the possibility of continuing to add new capabilities to your fleet connected devices, even after being deployed. It is all possible via releasing firmware.

As a product creator, you can choose to release firmware to *some* or *all* of your product fleet. Releasing a firmware as the **product default** sets the firmware as the default version available to *all* devices in the fleet to download and run.

To start the release process, place your cursor over the firmware you want to release and click **Release firmware**:



A modal will appear, asking you to confirm the action you are about to take:



Always confirm the version, targeted group(s) and impacted devices before releasing firmware to your device fleet to ensure you are taking the desired action.

Impacted devices refers specifically to the number of devices that will receive an OTA firmware update as a direct result of this action. Keep in mind that releasing firmware always presents risk. Anytime the code on a device is changed, there is a chance of introducing bugs or regressions. As a safeguard, **a firmware version must be successfully running on at least one device before it can be released**.

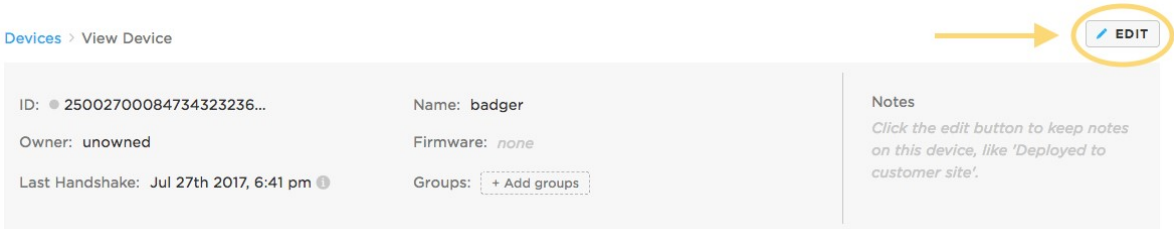
When you have confirmed the release is what you have intended, click the **Release this firmware** button. Note that the devices will not receive the firmware immediately; instead, they will be targeted for an over-the-air update the next time they start a new secure session with the cloud (this is called a *handshake*).

It is also possible to release firmware to a subset of your product fleet, using device groups. For more information on fine-grained firmware management, please check out the guide on device groups. Note that this is a feature currently only available to enterprise customers. Please contact us for access to this feature.

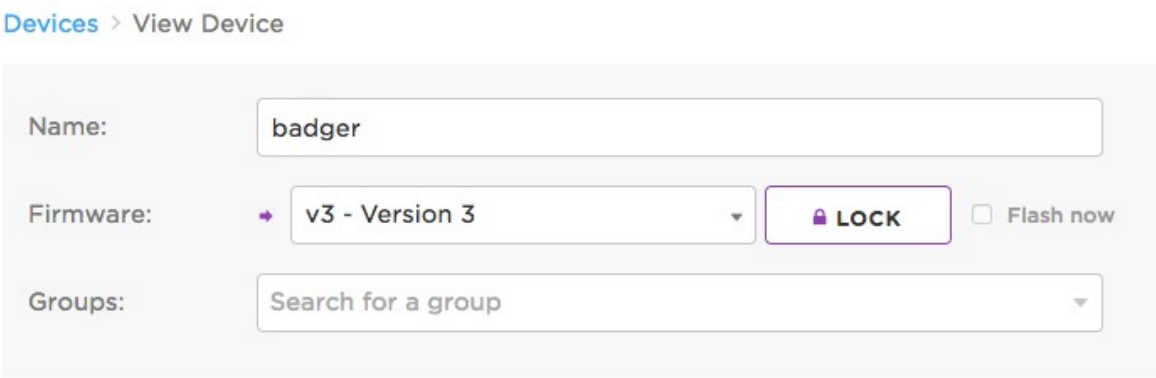
Locking firmware

In many cases, you may want to force a device to download and run a specific version of product firmware. This is referred to as **locking** the device. You can lock a device to a new version of product firmware to test it before releasing the firmware to the fleet.

To lock a device to a firmware, find the device on your product's devices view . Click on the device, which will take you to the device details view. Click on the **Edit** button:



This will allow you to edit many aspects of the device's state, including the firmware it is targeted to run. Find the **Firmware** section, select a version of firmware you want to lock the device to, and click the **Lock** button as sown below:



If the device is currently online, you can optionally immediately trigger an OTA update to the device by checking *Flash now* next to the lock button. Otherwise, the device will download and run the locked firmware the next time it handshakes with the cloud (starts a new secure session, most often on reset).

Once the device downloads and runs the locked firmware, it will no longer be targeted by the Particle cloud for automatic firmware updates, until it is unlocked. For more details, please read the firmware precedence rules.

Unlocking firmware

Unlocking a product device breaks its association with the locked firmware version and makes the device eligible to receive released product firmwares once again.

To unlock a devcie, visit the device's details view by clicking on it from your product's device list. Click the **Edit button** (shown above), and then click the **Unlock** button:

The device above is now unlocked from version 3 of product firmware, and may be targeted to receive a released firmware next time it handshakes with the cloud.

Firmware Precedence Rules

Devices in your fleet will be targeted to receive a version of product firmware according to these precedence rules:

- A device that has been **individually locked** to a version of product firmware is respected above all else, and will not be overwritten by any released firmwares.
- If unlocked, devices **belonging to a group** will receive the corresponding group's released firmware (if a firmware has been released to the group). When a device belongs to multiple groups that each have released firmware, the *highest firmware version* will be preferred
- If a device is unlocked and **does not belong to any groups** with released firmware, it will receive the **Product default** released firmware (if a firmware has been released as the Product default)
- If none of the above conditions result in a device being targeted for a product firmware, it will not receive an automatic OTA update from the Particle cloud

Managing Customers

Now that you have set up a Product, your customers will be able to create accounts on the Particle platform that are registered to your Product. When properly implemented, your customers will have no idea that Particle is behind the scenes; they will feel like they are creating an account with *ACME, Inc.*.

There are three ways you can authenticate your customers:

- **Simple authentication**. Your customers will create an account with Particle that is registered to your product. You do not need to set up your own authentication system, and will hit the Particle API directly.
- **Two-legged authentication**. Your customers will create an account on your servers using your own authentication system, and your web servers will create an account with Particle for each customer that is paired to that customer. Your servers will request a scoped access token for each customer to interact with their device. This is a completely white-labeled solution.
- **Login with Particle**. Your customers will create a Particle account and a separate account on your website, and link the two together using OAuth 2.0. Unlike the other authentication options, this option will showcase Particle branding. This is most useful when the customer is aware of Particle and may be using Particle's development tools with the product.

When you create your Product in the Console, you will be asked which authentication method you want to use. Implementation of these methods are covered in detail in the How to build a web app section of this guide.

As customers are created for your product, they will begin to appear on your Customers () page. For each customer, you will be able to see their username and associated device ID. Note that the device ID column will not be populated until the customer goes through the claiming process with their device.

Monitoring Event Logs

The Logs page () is also available to product creators! Featuring the same interface as what you are used to with the developer version of the Console, the logs will now include events from any device identifying as your product. Use this page to get a real-time look into what is happening with your devices. In order to take full advantage of the Logs page, be sure to use `Particle.publish()` in your firmware.

Managing your billing

To see all billing related information, you can click on the billing icon in your Product's sidebar (💳). This is the hub for all billing-related information and

actions. For more specifics about the pricing tiers and frequently asked questions, go check out the Pricing page.

How billing works

Product owners

Each Product has one primary administrator, the product owner. The owner manages the payment options, the pricing tier, and what billing interval the Product is on. This owner is the account that created the Product in the first place.

Devices

Devices are any physical device that uses the Particle Cloud- Photons, Electrons, P1s, P0s, etc. The only devices that count toward your pricing tier are those in a single Product fleet, so you could have many "loose" devices in your personal account without them adding to the number of devices in one of your Products.

Your personal account's devices (as well as new, unclaimed devices) are added to a Product when you want to use them as a group for data reporting, firmware updates, and collaboration with coworkers or friends. Only when they've been intentionally added to a Product will they count towards that total.

Events

An event is any single outbound message from your fleet that exits the Particle Cloud. For example, this could include a Particle.publish() call on a device that triggers a webhook, an integration such as IFTTT, or a server-sent event (SSE).

If you have multiple JS applications subscribed to an event then each will consume one of your outbound messages with each published event- watch out for running many copies of your app!

When you hit the outbound event limit you can choose to upgrade to the new tier or stay at the current tier. If you do choose to stay with the current tier then any future events that month won't be sent. You'll get a fresh batch of events at the beginning of the next month, and normal behavior will resume then. Don't

worry, we'll send you a warning before and an alert when you reach the event limit.

Team Members

Team members are people that you've added to a Product to give them administrative access to the data, controlling firmware on devices, adding integrations, and more. The product owner can add and remove team members as necessary, but they'll need to have a Particle account before they can be invited to join your team.

Billing periods

Products in the Console are billed separately, so if you have more than one they can be on different pricing tiers and have different billing dates. The billing date for a Product is the anniversary of when it was first created, either day of the month if you're on a monthly plan or the day of the year (month+day) in the case of an annually paid plan.

All Console plans are billed at the beginning of the plan period; in other words, it's prepaid. You'll pay for the following month or year, and can later cancel for a prorated remainder.

Status

It's easy to find out the status of your Product's metrics. Visit console.particle.io/billing and you'll see an up-to-date list of each Product you own, how many outbound events they've used that billing cycle, number of devices in each, and how many team members they have. The renewal date for each Product plan is also shown, so you know when the next bill is coming up.

Changing Plans

You can upgrade to a higher tier at any time from the Billing view in order to add more team members or devices, or increase your monthly outbound event limit. We'll send you notifications before you encounter the outbound event limit, but it's wise to upgrade early because no events will be sent after you hit it. If you've hit the event limit, tried to add more devices than are supported by the current

tier, or attempted to add a 6th team member while on the free Prototype tier, you'll be prompted to upgrade to a higher tier.

When you upgrade we'll prorate the new price for the rest of the month (or year, depending on your billing period) so you'll never be double-billed.

Contact us if you need to downgrade or remove a Product; both have potentially complex impacts and ambiguities, so we'll make sure that exactly the right thing happens.

Updating your credit card

You can update your credit card from the billing page by clicking on the "UPDATE" button next to the credit card on file. Whenever your credit card on file expires or no longer is valid, be sure to update your credit card info here to avoid any disruptions in your service.

Failed Payments

If we attempt to charge your credit card and it fails, we do not immediately prevent you or your team from accessing your Device Management Console. We will continue to retry charging your card once every few days **for a maximum of 3 retries**. You will receive an email notification each time an attempt is made and fails. When you receive this notification, the best thing to avoid any interruption in service is to update your credit card.

After we have unsuccessfully tried to charge your card 3 times, your Console account will be locked. **Your Products and all data will not be deleted**. After re-activating your account, you will be able to access your Console once again.

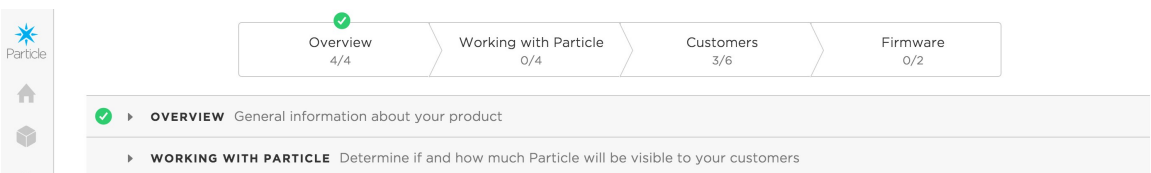
Configuring Your Product

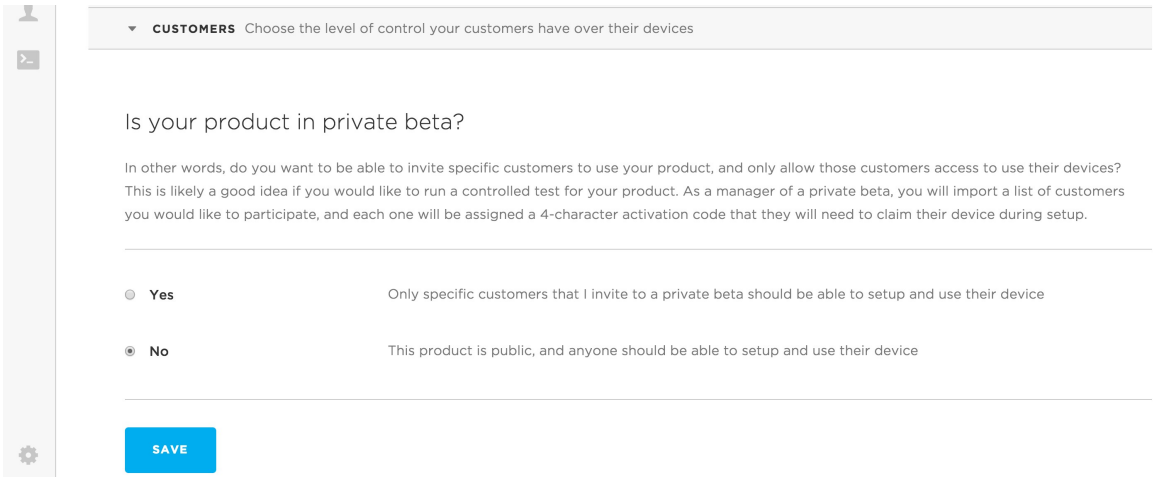
As a product creator, there are some key decisions you will need to make before devices are shipped to customers. Your configuration page will walk you through key questions that you should be thinking about during the development process. **You don't need to know the answers to all of these questions right now.** You are always able to return to your Configuration page to answer outstanding questions, or change your answers. However, you **should** answer all questions before you can start manufacturing.

It's also worth mentioning that some of the questions asked on the configuration page have tangible impacts on how your product will function within the Particle ecosystem, and others are simply educational to encourage you to be thinking strategically about what needs to happen before your product goes to manufacturing.

There are four main sections to the configuration page: *Overview*, *Working with Particle*, *Customers*, and *Firmware*. A few questions to highlight here:

- **Authentication/Logging in with Particle**: Thinking about how you would like to handle authentication is one of the earlier decisions you should be make as a product creator. There are three options for authentication: *simple auth*, *two-legged auth*, and *login with Particle (OAuth)*. Each option is explained in detail later. Picking an authentication method will likely depend on whether/how much you would like Particle to be hidden from your customers, as well as your development's team appetite for complexity.
- **Private Beta**: Do you only want a select group of people to use your product, inviting them as part of a private beta? This is likely a good idea if you would like to run a controlled test for your product. As a manager of a private beta, you will import a list of customers you would like to participate, and each one will be assigned a 4-character activation code that they will need to claim their device during setup.
- **Programming the product during manufacturing**: You can either program each device while they are on the manufacturing line, or send an OTA update to the device on customer setup. The main advantage of programming on the line is that the device will function immediately, instead of requiring the customer to be in range of Wi-Fi when they unbox the device. However, programming on the line will require your binary to be locked down and finalized before manufacturing begins. Programming the device on setup will allow you to continue developing the firmware for your product in between manufacturing and customer unboxing, providing additional flexibility. But, the device will not function properly until the customer connects the device to the internet and receives the OTA.





The configuration page will identify key decisions you will need to make before manufacturing

FIRMWARE MANAGER

The Firmware Manager is a desktop application that upgrades your Photon to the latest system firmware. It provides an easy way to update system firmware while avoiding cellular data charges.

Getting Started

The Firmware Manager is available for Windows and OS X.

Windows

Click Firmware Manager for Windows to download the application to your downloads folder. The file name will begin with "particle_firmware_manager".

Once the download is complete, double-click the downloaded file to run.

The first time the utility is run, you will be prompted to enable administrator access. This is so it can install the Particle USB Drivers to your machine.

OS X

Click Firmware Manager for OS X to download the application to your downloads folder.

The application is provided as a Zip file. Once the file has downloaded, double-click the file to start unpacking the application. This will take just a few seconds and you'll then see a folder named "Particle firmware manager" with a Particle icon.

Double-click the "Particle firmware manager" folder to launch the application.

Usage overview

Using the firmware manager follows these steps:

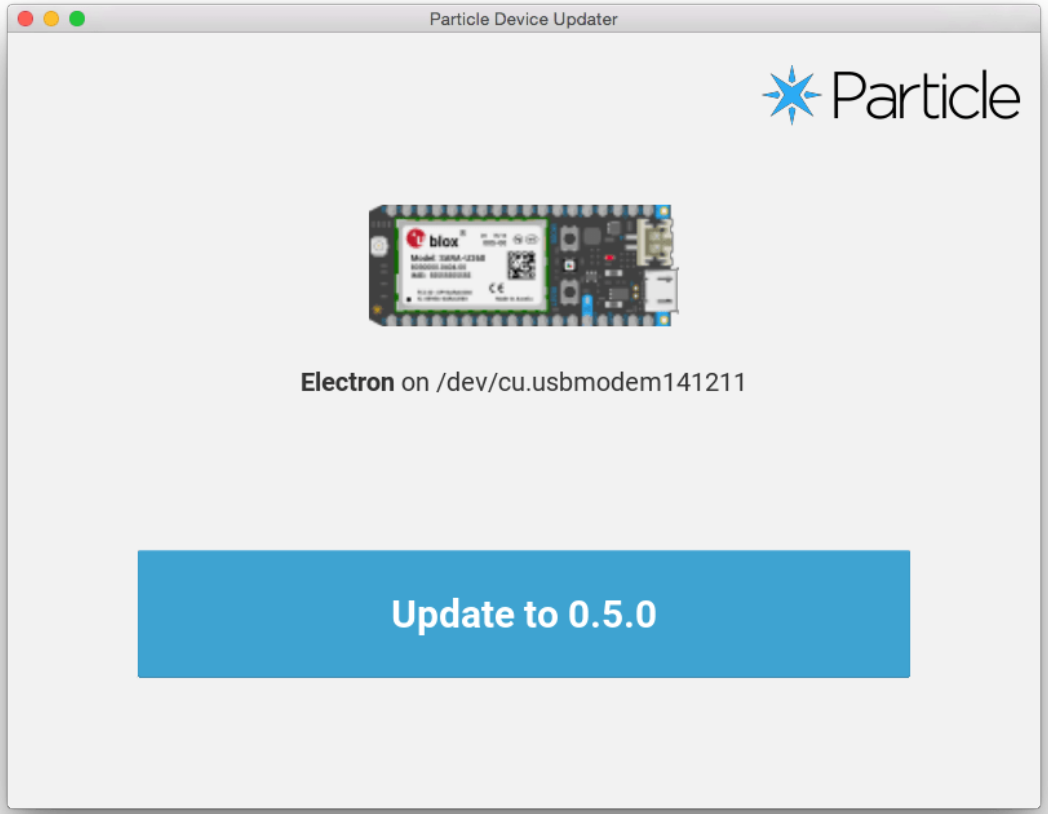
1. Launching the application
2. Connecting your device
3. Updating your device
4. Update complete!

Opening Screen

When the application starts, you'll see a screen like this:

Connecting your Device

If you haven't already connected your Photon to your computer with a USB cable, do that now. When connected, the screen will look like this:

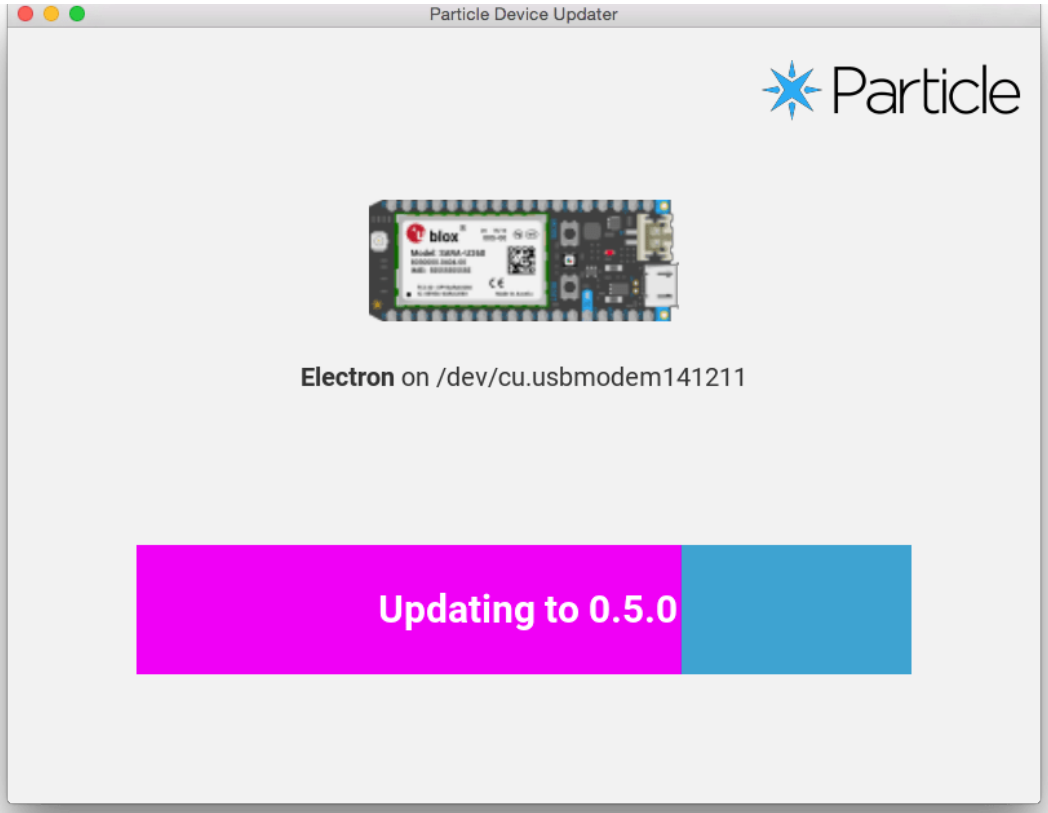


(On Windows, the device location will be shown as COM1: or similar.)

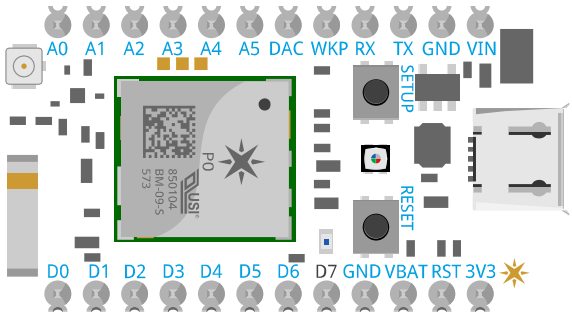
Updating Firmware on your Device

When you're ready to update your device, click on the button "Update to 0.5.0". (The version number may be different.)

The progress bar will start to change to magenta as the update progresses



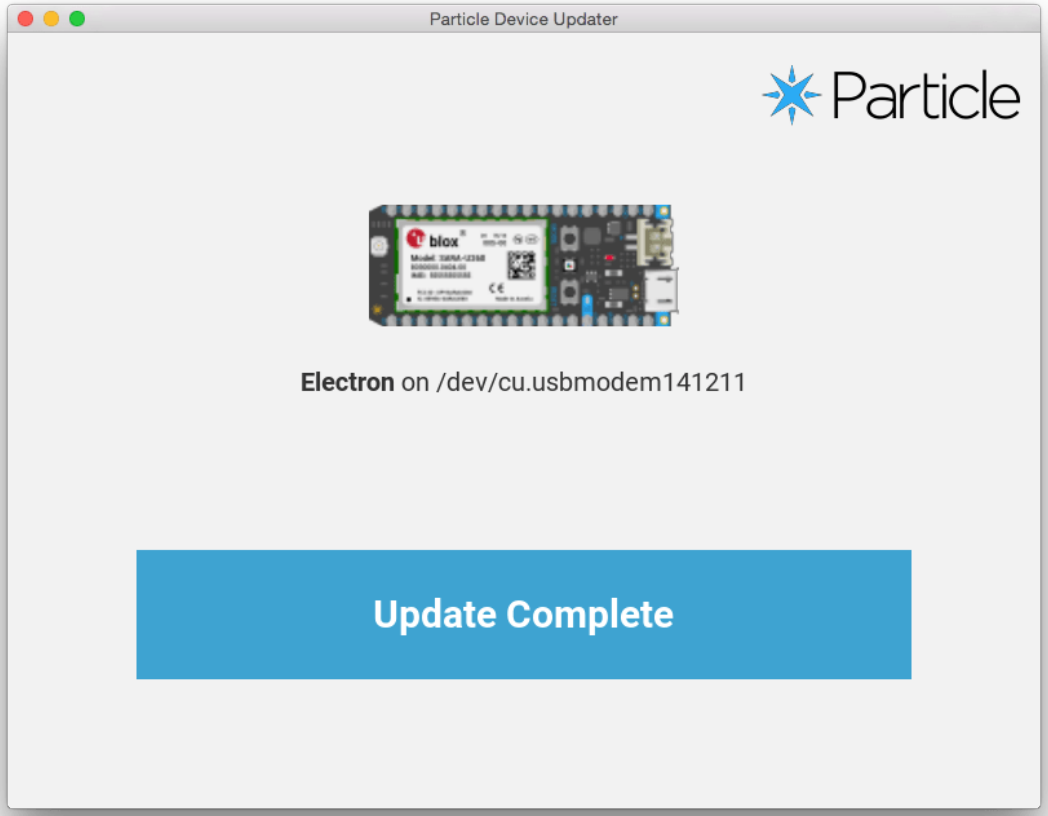
... and the LED on your device will start blinking magenta, like this:



The update will pause for a short while halfway through as the device restarts and gets ready for the second part of the update. You don't need to do anything here - the device will automatically continue with the next part of the update.

Update complete

When the update is complete you'll see this screen



Congratulations! You've updated the system firmware on your device.

Your device will automatically reboot and will run your app using the updated system firmware.

Troubleshooting

Device not detected

If the application doesn't detect your device, try disconnecting and reconnecting the device. This is especially true on Windows after installing the USB drivers - Windows will not detect the device using the newly installed drivers until the device has been disconnected and reconnected.

Update stops halfway

If the update gets stuck or you see an error message, try following these steps

- Close the device updater if it's already open
- Connect the device directly to your computer rather than via USB hubs
- Restart the device in Safe Mode
- When the device has connected to the cloud, put the device in listening mode
- launch the device updater and perform the update as described above

We hope that fixes it for you, but if you are still experiencing problems, please check our GitHub Issues and file a new issue.

PARTICLE CLI

The Particle CLI is a powerful tool for interacting with your devices and the Particle Cloud. The CLI uses Node.js and can easily run on Windows, Mac OS X, and Linux. It's also open source so you can edit and change it, and even send in your changes as pull requests if you want to share!

Installing

Using macOS or Linux

The easiest way to install the CLI is to open a Terminal and type:

```
bash <( curl -sL https://particle.io/install-cli )
```

This command downloads the `particle` command to your home directory at `~/bin`, installs a version of Node.js to `~/.particle` and installs the `particle-cli` Node.js module that contain the code of the CLI.

It will also try to install DFU-util, a utility program for programming devices over USB. See the instructions for installing DFU-util if the installer is not able to automatically install dfu-util.

The installer also works on the Raspberry Pi!

Using Windows

Download the Windows CLI Installer and run it to install the Particle CLI, the device drivers and DFU-util.

The CLI is installed to `%LOCALAPPDATA%\particle` (`C:\Users\username\AppData\Local\particle` for Windodws in English).

Advanced Install

You can manually install the `particle-cli` Node.js package if you need the CLI installed in a different location or you need to install a specific version of the CLI.

Make sure you have a recent LTS version of Node.js installed.

```
# check that you have node.js 6 or above. Check http://nodejs.org
on how to update node.js
$ node -v
v6.11.4
# check that you have npm 5 or above
$ npm -v
5.5.1
# update npm if your version is older than 5
$ npm install -g npm
```

Next, open a command prompt or terminal, and install by typing:

```
# how to install the particle-cli
$ npm install -g particle-cli
$ particle login
```

If you experience permission errors, we recommend you change the directory where npm installs global packages (ones installed with `-g`) to another directory as documented here. If you must install `particle-cli` to the default global package location as the superuser, you have to use the `--unsafe-perm` flag to successfully install all dependencies: `sudo npm install -g --unsafe-perm particle-cli`.

For more OS-specific install instructions, see below.

To use the local flash and key features you'll also need to install dfu-util, and openssl. They are freely available and open-source, and there are installers and binaries for most major platforms.

Here are some great tutorials on the community for full installs:

Install Separate Components for Windows

Installing on Ubuntu 12.04

Installing on Ubuntu 14.04

Installing on Mac

Upgrading to the latest version

If you installed the Particle CLI through the installer, it will periodically update itself to the latest version.

To force it to update, run the installer script again or enter this command:

```
# how to update the installed CLI
$ particle update-cli
```

If you installed manually using `npm install`, you can upgrade by running the same command you used to install the tool.

Running from source (advanced)

To grab the CLI source and play with it locally

```
# how to get the source code for the CLI
$ git clone https://github.com/particle-iot/particle-cli.git
$ cd particle-cli
$ npm install
$ node bin/particle.js help
```

Getting Started

These next two commands are all you need to get started setting up an account, claiming a device, and discovering new features.

particle setup

This command will guide you through logging in or creating a new account as well as claiming your device!

```
# how to setup your account and your device
$ particle setup
```

particle help

Shows you what commands are available and how to use them. You can also give the name of a command for detailed help.

```
# how to get help
$ particle help
$ particle help keys
```

Flashing over Serial for the Electron

If you're wanting to save data on your Electron you should definitely consider flashing your Electron over serial instead of OTA (over-the-air).

Assuming you've compiled and downloaded the firmware binary from Build IDE by clicking the cloud button next to the file name, you should be able to use the Particle CLI, mentioned above, to flash your application firmware to your Electron *without using data.*

Steps:

- **1:** Put the Electron in to listening mode
- **2:** Open Particle CLI from your machine (make sure you've installed it first)
- **3:** Navigate to the folder where you've downloaded the `firmware.bin` file.
- **4:** From the CLI issue `particle flash --serial firmware.bin`

```
# How to flash an Electron over serial
$ particle flash --serial firmware.bin

! PROTIP: Hold the SETUP button on your device until it blinks
blue!
? Press ENTER when your device is blinking BLUE
sending file: firmware.bin

Flash success!
```

Note: If your Electron goes into safe mode, this is okay, just make sure that the system firmware you on the device matches the dependency of the system firmware built into the firmware application.

Blink an LED with Tinker

If you're just opening a new device, chances are it's already loaded with Tinker, the app we load at the factory. If you don't have Tinker, or if you've been using the build IDE already, let's load it quickly by typing:

```
# How to re-load tinker onto a device
$ particle flash my_new_device_name tinker
Including:
/usr/local/lib/node_modules/particle-cli/binaries/particle_tinker.b
in
attempting to flash firmware to your device my_new_device_name
flash device said  {"id":"0123456789ABCDEFGHI","status":"Update
started"}
```

Let's make sure your device is online and loaded with Tinker. We should see the four characteristic functions exposed by Tinker, "digitalWrite", "digitalRead", "analogWrite", and "analogRead".

```
# how to show all your devices and their functions and variables
$ particle list

Checking with the cloud...
Retrieving devices... (this might take a few seconds)
```

```
my_device_name (0123456789ABCDEFGHI) 0 variables, and 4 functions
  Functions:
    int digitalread(String args)
    int digitalwrite(String args)
    int analogread(String args)
    int analogwrite(String args)
```

Let's try turning on the LED attached to pin D7 on your device.

```
# how to call a function on your device
$ particle call my_device_name digitalwrite D7,HIGH
1
$ particle call my_device_name digitalwrite D7,LOW
1
```

Nice! You should have seen the small blue LED turn on, and then off.

Update your device remotely

You can write whole apps and flash them remotely from the command line just as you would from the build IDE. Let's write a small blink sketch to try it out.

Copy and paste the following program into a file called blinky.ino

```
// Copy me to blinky.ino
#define PIN D7
int state = 0;

void setup() {
    //tell the device we want to write to this pin
    pinMode(PIN, OUTPUT);
}
void loop() {
    //alternate the PIN between high and low
    digitalWrite(PIN, (state) ? HIGH : LOW);

    //invert the state
    state = !state;

    //wait half a second
```

```
    delay(500);
}
```

Then let's compile that program to make sure it's valid code. The CLI will automatically download the compiled binary of your program if everything went well.

```
# how to compile a program without flashing to your device
$ particle compile photon blinky.ino
Including:
blinky.ino
attempting to compile firmware
pushing file: blinky.ino
grabbing binary from:
https://api.particle.io/v1/binaries/01234567890ABCDEFGH
saved firmware to firmware_123456781234.bin
Compiled firmware downloaded.
```

Now that we have a valid program, let's flash it to our device! We can use either the source code again, or we can send our binary.

```
# how to flash a program to your device (from source code)
$ particle flash my_device_name blinky.ino

# OR - how to flash a pre-compiled binary to your device
$ particle flash my_device_name firmware_123456781234.bin
Including:
firmware_123456781234.bin
attempting to flash firmware to your device my_device_name
flash device said  {"id":"01234567890ABCDEFGH","status":"Update
started"}
```

Compile and flash code locally

You can find a step-by-step guide to installing the local build toolchain for the firmware in the FAQ section of the documentation.

After building your code on your machine, you can flash it to your device over Serial or remotely.

Working with projects and libraries

When your code gets too long for one file or you want to use libraries that other developers have contributed to the Particle platform it's time to create a project.

Creating a project

By default projects are created in your home directory under Particle or in your Documents folder under Particle on Windows. You can also create projects in the current directory.

```
$ particle project create
What would you like to call your project? [myproject]: doorbell
Would you like to create your project in the default project
directory? [Y/n]:
Initializing project in directory
/home/user/Particle/projects/doorbell...
> A new project has been initialized in directory
/home/user/Particle/projects/doorbell
```

Using libraries

The CLI supports using libraries with your project. This allows you to incorporate already written and tested code into your project, speeding up development and assuring quality.

The overall flow when consuming a library goes like this

- set up the initial project for your application
- find the library you want to add `particle library search`
- add the library to your project - `particle library add`
- edit your source code to use the library
- compile your project - `particle compile`

These commands are described in more details in the CLI reference.

library search

The `library search` command allows you to search for libraries that are related to the text that you type in.

For example,

```
particle library search neo
```

Will find libraries containing `neo` in their name.

library add

The `library add` command adds the latest version of a library to your project.

For example, if you wanted to add the InternetButton library to your project, you would type

```
$ particle library add internetbutton
> Library InternetButton 0.1.10 has been added to the project.
> To get started using this library, run particle library view
InternetButton to view the library documentation and sources.
```

This will add the InternetButton dependency to your project's `project.properties` file.

The InternetButton library itself is not present in your project, so you won't see the InternetButton sources. The library is added to your project when the project is compiled in the cloud.

To make the library functionality available to your application, you add an include statement to your application source code. The include statement names the library header file, which is the library name with a `.h` ending.

For example, if we were using the library "UberSensor", it would be included like this

```
#include "UberSensor.h"
```

library view

The `library view` downloads the source code of a library so you can view the code, example and README.

```
$ particle library view internetbutton
Checking library internetbutton...
Library InternetButton 0.1.11 installed.
Checking library neopixel...
Checking library particle_ADXL362...
Library particle_ADXL362 0.0.1 installed.
Library neopixel 0.0.10 installed.
To view the library documentation and sources directly, please
change to the directory
/home/monkbroc/Particle/community/libraries/InternetButton@0.1.11
```

Change to the directory indicated to view the sources.

library copy

Adding a library to your project does not add the library sources. For times when you want to modify the library sources, you can have them added locally.

```
particle library copy neopixel
```

The library will be copied to the `lib` folder of your project. If you already have the library in your `project.properties` make sure to remove it so the cloud compiler doesn't overwrite your changed copy with the published code.

Incorporating the library into your project

Once the library is added, it is available for use within your project. The first step to using the library is to include the library header, which follows the name of the library. For example:

```
#include "neopixel.h"
```

The functions and classes from that library are then available for use in your application. Check out the library examples and documentation that comes with the library for specifics on using that library.

Contributing Libraries

Contributing a library is the process where you author a library and share this with the community.

The steps to creating a library are as follows:

- optionally, create a project for consuming the library
- scaffold a new library structure - `library create`
- author the library, tests and examples
- publish the library

Create a project for consuming the library

While it's not strictly necessary to have a project present when authoring a new library, having one can help ensure that the library works as intended before publishing it. The project allows you to consume the library, check that it compiles and verify it behaves as expected on the target platforms before publishing.

For the library consumer project that will consume the library `mylib`, create an initial project structure that looks like this:

```
src/project.cpp
src/project.h
project.properties
lib/mylib
```

The library will exist in the directory `lib/mylib`.

All these files are initially empty - we'll add content to them as the library is authored.

Scaffolding the library

The `library create` command is used to scaffold the library. It creates a skeleton structure for the library, containing initial sources, examples, tests and documentation.

In our example project structure we want to create a new library in `lib/mylib` so we will run these commands:

```
cd lib/mylib
particle library create
```

The command will prompt you to enter the name of the library - `mylib`, the version - `0.0.1` and the author, your name/handle/ident.

The command will then create the skeleton structure for the library.

Authoring the library

You are then free to edit the `.cpp` and `.h` files in the `lib/mylib/src` folder to provide the functionality of your library.

It's a good idea to test often, by writing code in the consuming project that uses each piece of functionality in the library as it's written.

Consuming the library

To test your changes in the library, compile the project using `particle compile <platform>`

```
particle compile photon
```

This will create a `.bin` file which you then flash to your device.

```
particle flash mydevice firmware.bin
```

(Replace the name `firmware.bin` with the name of the `.bin` file produced by the compile step.)

Contributing the library

Once you have tested the library and you are ready to upload the library to the cloud, you run the `library contribute` command. You run this command from the directory containing the library

```
cd lib/mylib
particle library contribute
```

Before the library is contributed, it is first validated. If validation succeeds, the library is contributed and is then available for use in your other projects. The library is not available to anyone else.

Publishing the Library

If you wish to make a contributed library available to everyone, it first needs to be published.

When publishing a library, it's important to ensure the version number hasn't been published before - if the version has already been published, the library will not be published and an error message will be displayed.

Incrementing the version number with each publish is a recommended approach to ensuring unique versions.

Once the library is published, it is visible to everyone and available for use. Once the a given version of a library has been published, the files and data cannot be changed. Subsequent changes must be via a new contributed version and subsequent publish.

Reference

For more info on CLI commands, go here.

Also, check out and join our community forums for advanced help, tutorials, and troubleshooting.

Go to Community Forums >

PARTICLE CHANNEL ON IFTTT

Introduction

Anything you build with Particle is now easily available on IFTTT! IFTTT is a service that lets you create powerful connections with one simple statement: "If this then that." There is a great introduction to their ecosystem on IFTTT's website. Go check it out if you haven't already, it's super helpful.

The Particle Channel on IFTTT will let you connect your devices to other powerful channels. You can now easily send and receive tweets, SMS, check the weather, respond to price changes, monitor astronauts, and much, much more. This page is a reference for you to use as you get your Particle Recipes set up.

If you're totally new to Particle, that's okay! Before you get going on the Particle channel, be sure to get your Photon or Core connected and claimed. Get started here.

Lets go!

Parts of an IFTTT Recipe

What are triggers?

Triggers are the *this* part of a Recipe. Triggers are how IFTTT knows when to run your recipe. A trigger can be as conceptually simple as "is x greater than 5?" or "did I get new email?", and they frequently are!

What are actions?

Actions are what IFTTT does when the answer to your trigger question is "yes!" When set up, you can have IFTTT email you, post for you, save info to Dropbox, or many other useful functions.

What are Recipes?

A combination of a Trigger and an Action. IFTTT lets you connect triggers to actions. Did someone tweet something interesting (that's a trigger), then turn my disco ball on (that's an action).

What are ingredients?

Ingredients are pieces of data from a Trigger. Ingredient values are automatically found by IFTTT using certain aspects of your device and/or firmware. These pieces of data can be used when setting up the Action that goes with your created Trigger. For Particle, ingredients will often include the name you've given your Core or Photon, the time that the trigger occurred, and any data that trigger returned.

Other IFTTT channels will provide (and sometimes automatically insert) their own ingredients. If ingredients are available, they can be found in the blue Erlenmeyer flask icon next to the relevant input box.

Before you build with Particle + IFTTT

Firmware is key: IFTTT will pull directly from the firmware that is currently flashed to your devices. It will only show functions,variables,etc from firmware that is currently flashed to one of your devices. That means that if, for example, you're trying to use the "Monitor a Function" Trigger you'll need to have flashed firmware to your board that includes Particle.function().

But what if I want to try this without writing firmware? We recommend starting with the Monitor a Device Status Trigger. You can use this Trigger with the firmware that came with your device.

Okay, I'm ready to build my own firmware. Where do I start? Get to know the web IDE and explore the Particle community site for great tutorials, examples and advice on projects.

Triggers

New Event Published

```
// SIMPLEST SYNTAX
Particle.publish(String eventName);

// EXAMPLE DEVICE CODE
Particle.publish("Boiling!");


// SYNTAX FOR SENDING DATA
Particle.publish(String eventName, String data);

// EXAMPLE DEVICE CODE
Particle.publish("Boiling!", "212");


// THE COMPLETE VERSION, useful for making events private
Particle.publish(String eventName, String data, int ttl, PRIVATE);

// EXAMPLE DEVICE CODE
Particle.publish("Boiling!", "212", 60, PRIVATE);
```

Firmware requirements

To use this Trigger, firmware must include Particle.publish(). Complete documentation on using Particle.publish() is here.

A word of caution - firmware loops quickly, so it's very easy to run publish() too frequently. You'll trigger your IFTTT recipe 100 times in a blink, and if you publish() more than once a second then the Particle Cloud will briefly disable further publishes. Make sure to think through the logic of your code so that it only publishes when you actually want it to.

Trigger fields

If (Event Name): The name you gave to Particle.publish() in your firmware. Also referred to as topic, name or channel.

is (Event Contents): *Optional* Any data you included with your Particle.publish call from firmware.

Device Name or ID: This dropdown menu will be automatically populated with the names of Particle devices that are claimed to your account and are loaded with firmware.

Ingredients

EventName: The name you gave to Particle.publish("name") in your firmware.

```
// EXAMPLE EVENT NAME
allbuttons
```

EventContents: What data the event contained *optional*

```
// EXAMPLE CONTENTS
"on"
```

DeviceName: The name you provided when you claimed your Particle device.

```
// EXAMPLE DEVICE NAME
MyDevice
```

CreatedAt: Date and time the event was published

```
// EXAMPLE TIMESTAMP
January 12, 2015 at 6:59pm
```

Monitor a variable

Firmware requirements

Monitoring variables is also a simple way to get going. You'll need to create a variable at the top of your code, call Particle.variable() using the format to the right in the setup() function, and then you're good to go.

Complete documentation on using Particle.variable() is here.

```
// EXAMPLE SHOWING THREE DATA TYPES
int analogvalue = 0;
double tempC = 0;
char *message = "my name is particle";

void setup()
{
  // variable name max length is 12 characters long
  Particle.variable("analogvalue", &analogvalue, INT);
  Particle.variable("temp", &tempC, DOUBLE);
  Particle.variable("mess", message, STRING);
  pinMode(A0, INPUT);
}

void loop()
{
  // Read the analog value of the sensor (TMP36)
  analogvalue = analogRead(A0);
  //Convert the reading into degree celcius
  tempC = (((analogvalue * 3.3)/4095) - 0.5) * 100;
  delay(200);
}
```

Trigger fields

If (Variable Name): Select the Particle.variable you'd like to use from an automatically populated dropdown menu. These options will be pulled from the firmware flashed to your claimed devices. No options will be displayed if your devices don't have Particle.variable()s defined, so check your code and reflash your firmware if it's empty.

```
// EXAMPLE VARIABLE NAME
temperature
```

is (Test Operation): Select a comparison based on a dropdown menu with options: greater, less than, equals, greater or equal, less or equal

Comparison Value (Value to Test Against): The value that you are comparing with your Particle.variable().

```
// EXAMPLE VALUE
90
```

Ingredients

Value: The actual value of your variable. While your trigger will be doing a comparison, you can also use the value it returned in your action. This is useful for logging to email, Dropbox, or spreadsheets.

```
// EXAMPLE VALUE
72
```

Variable: The name of the Particle.variable that you are measuring.

```
// EXAMPLE VARIABLE NAME
temperature
```

DeviceName: The device the variable came from.

```
// EXAMPLE DEVICE NAME
MyDevice
```

CreatedAt: Date and time the event was created.

```
// EXAMPLE TIMESTAMP
January 17, 2015 at 7:52am
```

Monitor a function result

```
// SYNTAX TO REGISTER A PARTICLE FUNCTION
Particle.function("cloudNickname", firmwareFunctionName);
//                 ^
//                 |
//     (max of 12 characters long)

// EXAMPLE USAGE
int brewCoffee(String command);

void setup()
{
  // register the Particle function
  Particle.function("brew", brewCoffee);
}

void loop()
{
  // this loops forever, doing nothing
}

// this function automagically shows up in IFTTT
int brewCoffee(String command)
{
  // look for the matching argument "coffee" <-- max of 64
characters long
  if(command == "coffee")
  {
    //things you want it to do
    activateWaterHeater();
    activateWaterPump();

    //Returns the value "1" if it was successful
    return 1;
  }
  else return -1;
}
```

Your Particle.function()s can be used in several ways with IFTTT. You can send a value to your function to look up data, perform behaviors before taking a

measurement, or ask for something to happen and get a response when it's successful.

Firmware requirements

All of the details are covered in the example to the right. Just remember to declare a function at the top of your code, make it Particle.function() in setup(), and then declare what the function does down below. Currently, only the first 4 Particle.function()s that you register will show up in IFTTT.

Complete documentation on using Particle.function() is here.

Trigger fields

If the output value of (Function Name): The name of your function.

```
// EXAMPLE USAGE
allLedsOn()
```

when you send it (Value): *Optional* This is the data you're giving to the Particle.function() in your firmware.

```
// EXAMPLE USAGE, RGB VALUES FOR COLOR
100,100,100
```

is (Test Operation): Select a comparison based on a dropdown menu with options: greater, less than, equals, greater or equal, less or equal

Comparison Value: The value that you are comparing with your Particle.function result.

Ingredients

FunctionName: The function you just called.

```
// EXAMPLE USAGE
allLedsOn
```

FunctionResult: The number returned when you call the function. This may be a value from a sensor, or it could be a 0 or 1 which will confirm whether your function occurred.

```
// EXAMPLE USAGE
1
```

DeviceName: The device the variable came from.

```
// EXAMPLE USAGE
MyDevice
```

CreatedAt: Date and time the event was created.

```
// EXAMPLE TIMESTAMP
May 2, 2013 at 9:02am
```

Monitor your device status

Firmware requirements

You must have firmware on your Particle device, but nothing else is necessary. Basically, you're fine unless you've actively wiped your device.

Trigger fields

If (Device Name or ID: This dropdown menu will be automatically populated with the names of Particle devices that are claimed to your account and are loaded with firmware.

is (Status of your device): Selected from a dropdown menu - either online or offline.

Ingredients

DeviceName: The name of the device that changed status

```
// EXAMPLE USAGE
MyDevice
```

Status: Indicates whether your device is online or offline

```
// EXAMPLE USAGE
Offline
```

CreatedAt: Date and time that the status change occurred

```
// EXAMPLE TIMESTAMP
November 13, 2015 at 6:02pm
```

Actions

Publish an event

Firmware requirements

This action has IFTTT publishing an event, so your Particle device needs to subscribe to that event. These are complementary; a Particle.subscribe ("myEventName") watches for a publish("myEventName") and runs a function when it sees this matching event name. This means that even though this action is called *Publish* an event, your firmware needs to include Particle.subscribe().

```
int i = 0;

void myHandler(const char *event, const char *data)
{
  i++;
```

```
    Serial.print(i);
    Serial.print(event);
    Serial.print(", data: ");
    if (data)
    Serial.println(data);
    else
    Serial.println("NULL");
  }

  void setup()
  {
    Particle.subscribe("temperature", myHandler);
    Serial.begin(9600);
  }
```

To use `Particle.subscribe()`, define a handler function and register it in `setup()`.

You can listen to events published only by your own Cores by adding a `MY_DEVICES` constant.

```
  // only events from my Cores
  Particle.subscribe("the_event_prefix", theHandler, MY_DEVICES);
```

Complete documentation on using Particle.subscribe() is here.

Action fields

Then publish (Event Name): This may be autopopulated by the Trigger you have chosen, however you'll want to delete that and write in the Event Name that you have defined in your firmware. Also known as topic, event name, or channel.

```
  // EXAMPLE USAGE
  temperature
```

The event includes (Data): *Optional* The data that comes along with the event - a temperature value or sensor measurement for example.

is this a public or private event? Select either private or public.

Call a function

It's important to note that if you turn off the board that is attached to this action while your recipe is still live, the IFTTT servers may disable your recipe. This is easy to fix, just turn it back on again. Your default IFTTT settings are set up to send you an email when your recipe encounters a serious issue (like not having a device to run the requested function). You can always change these by clicking on your username at the top of the IFTTT menu and selecting "Preferences".

Firmware requirements

This is very similar to using a Particle.function() as a trigger, only you won't be using any values it returns. The same setup on the firmware side, and the example code above for Particle.function() as trigger, will work for this as well.

Complete documentation on using Particle.function() is here.

Action fields

Then call (Function Name): Select the function you'd like to use from the options in a dropdown menu. These options will be pulled from the Particle.function()s that are in the firmware flashed to any of your claimed Cores or Photons. IFTTT will list the first 4 function()s defined in your firmware.

```
// EXAMPLE USAGE
brew
```

with input (Function Input): This is an optional field. It may be automatically populated with ingredients from the trigger that you chose (i.e. Twitter may put in something like "Favorite tweet: TweetEmbedCode"). If you don't have specific code in your function() to use this data, it's best to delete it. There will be cases when you'll want to send some specific input.

FAQs

- How often should I expect IFTTT to check my triggers?

After you create a recipe we'll check functions or variables once a minute and then notify IFTTT right away if your trigger happens. Other channels will frequently take a while (up to 15 minutes) to trigger or perform their action.

- IFTTT says there is an error or something in my recipe log... what gives?

  Make sure your device is online and still running firmware that has the (event, function, variable, etc) that you linked in your recipe. If your device isn't connected when IFTTT tries to call a function, it won't work.

- I don't see my function in the list on IFTTT?

  Make sure you flashed your firmware to your device with the function you've exposed, and try refreshing the IFTTT page. You can confirm what functions are available using Particle Dev and clicking the 'Cloud variables and functions' menu or in the Particle-CLI by running `particle list`

- Why can I log in on build.particle.io, but I can't log in on IFTTT?

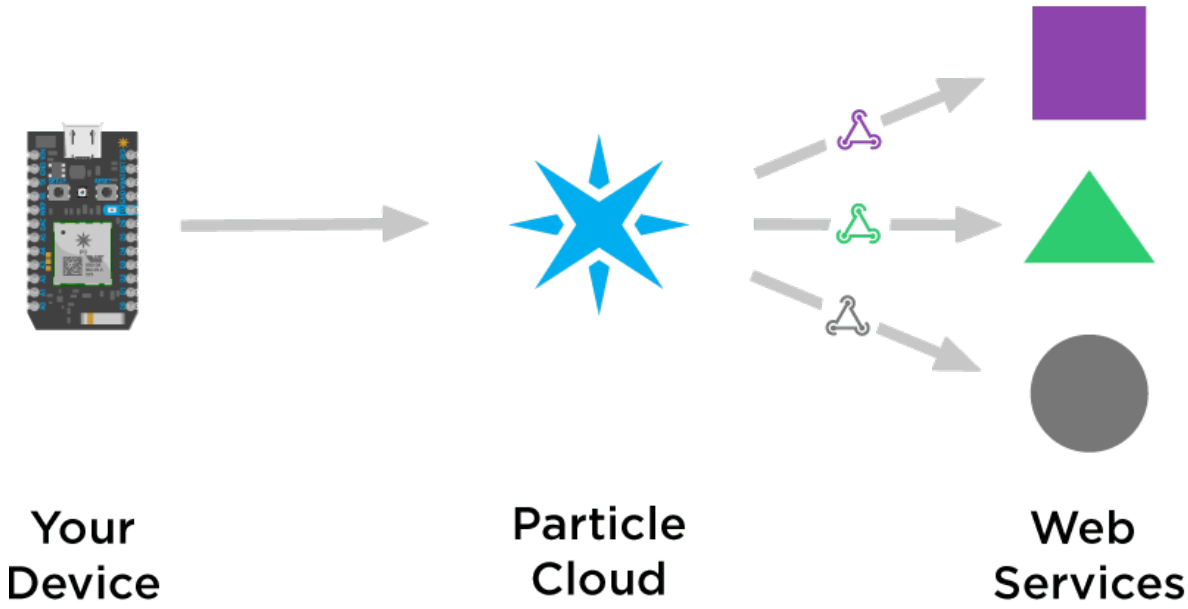
  Try just once more, and make sure that you're typing your username and password exactly how you entered it when creating your account. The build site is a little more forgiving than others at the moment.

Also, check out and join our community forums for advanced help, tutorials, and troubleshooting.

WEBHOOKS

Webhooks are a simple and flexible way to send data from your Particle devices to other apps and services around the Internet. Webhooks bridge the gap between the physical and the digital world, helping you get your data where you need it to be.

You could use a webhook to save valuable information in a database, visualize data being read from a sensor, send the latest weather report to your device, trigger a payment, send a text message, and so much more!



Webhooks allow you to send data from your connected device anywhere on the Internet

In this guide, we'll provide an overview of how you can use webhooks in your connected products, and walk you through a few examples to get you started.

If you are looking for all the juicy details, head over to the webhook reference page.

How webhooks work

Webhooks are tightly integrated with Particle's event system. Devices have the ability to both publish events to the Particle cloud, as well as subscribe to events from the cloud.

A webhook listens for a specific event published by a device. When this event is published, the webhook triggers a web request to a URL on the web. The request sent by the webhook can include information about the event, such as its name as well as any data included when the event was published.

You can configure a webhook to make different types of web requests. The most common type of webhook request is a `POST`, which is a method of *sending data* to another web server. In the case of Particle webhooks, this would mean sending data from your devices to a third-party web service. Other types of web requests, like `GET` and `PUT` can also be made with webhooks.

Often times, a web server you hit with a webhook will return data to you as a result of the request made. When this happens, your devices can subscribe to a specific event name to receive the response from the web server and use it in your firmware logic.

The combination of webhooks with the Particle cloud's pub/sub event system creates a very efficient way for you to leverage online tools and services and integrate them into your connected product.

Webhooks is one piece of a larger puzzle of Particle Integrations. We want to make it incredibly easy to send data from your devices wherever you need it. In the near future, Particle will offer branded integrations will further simplify the process of sending your data to useful web services.

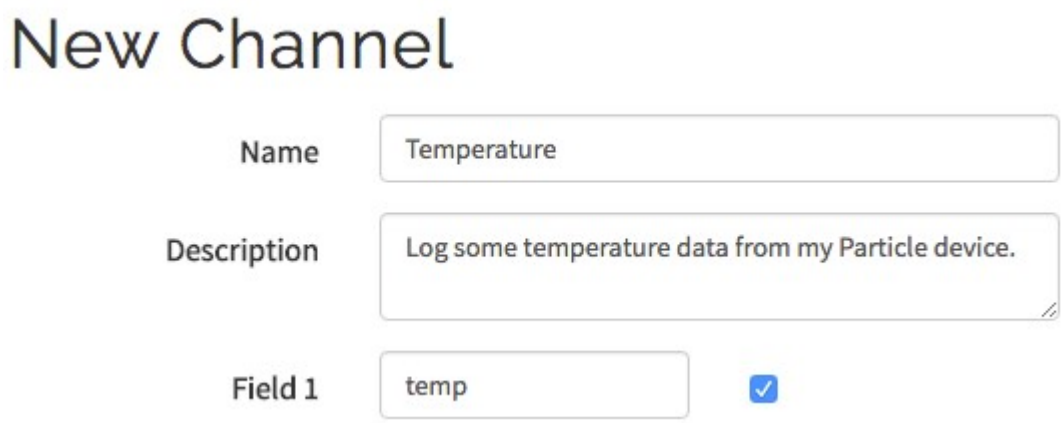
Your first webhook

Let's get started! For your first webhook, let's try to send some data from your Particle device to a graphing tool. For this example, we'll use ThingSpeak.

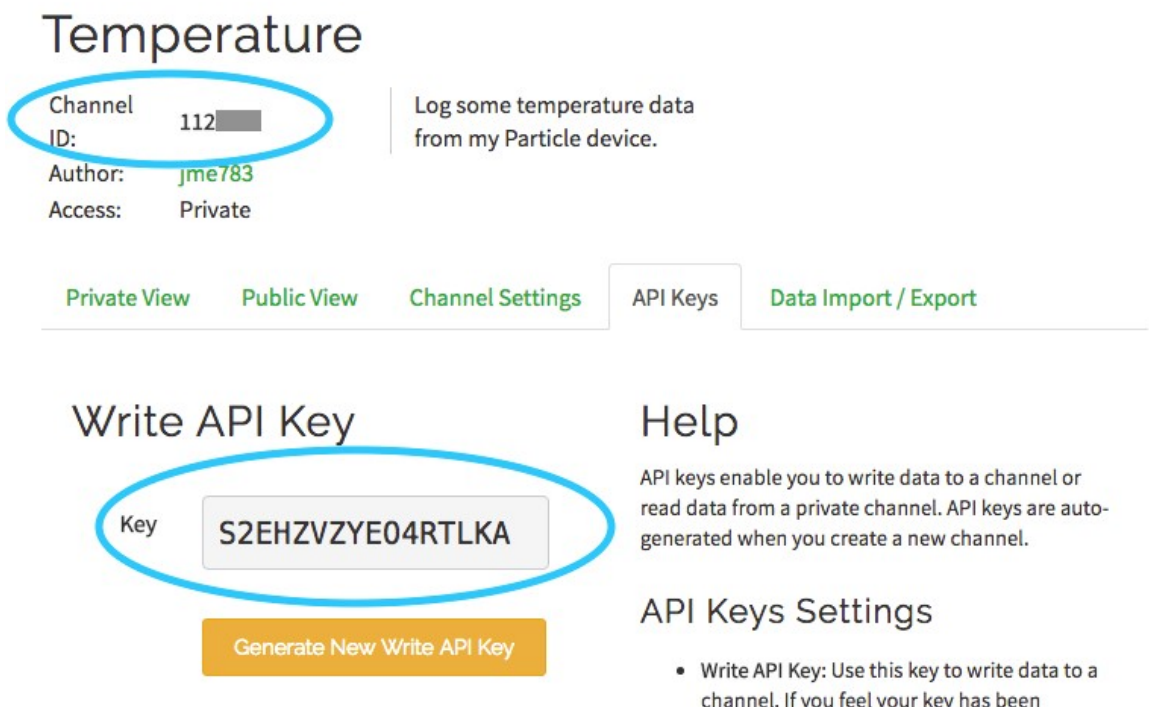
Configure ThingSpeak

Create a ThingSpeak account if you don't already have one. Next, create a channel by clicking the "New Channel" button on your ThingSpeak dashboard.

Name your channel "Temperature," add one field called "temp" and create the channel.



Once you've created the channel, you will need to note the Channel ID as well as your Write API key to use for creating the Particle webhook.



Great! We have what we need from ThingSpeak. Now let's go and create the webhook.

Create the webhook

The hub for managing your webhooks is the Particle Console. Log into your console and click on the Integrations tab. If you have created any webhooks in the past, they will appear here. Click on "New Integration" -> "Webhook" to get started.

Your Integrations Hub is where you can view and manage Particle webhooks

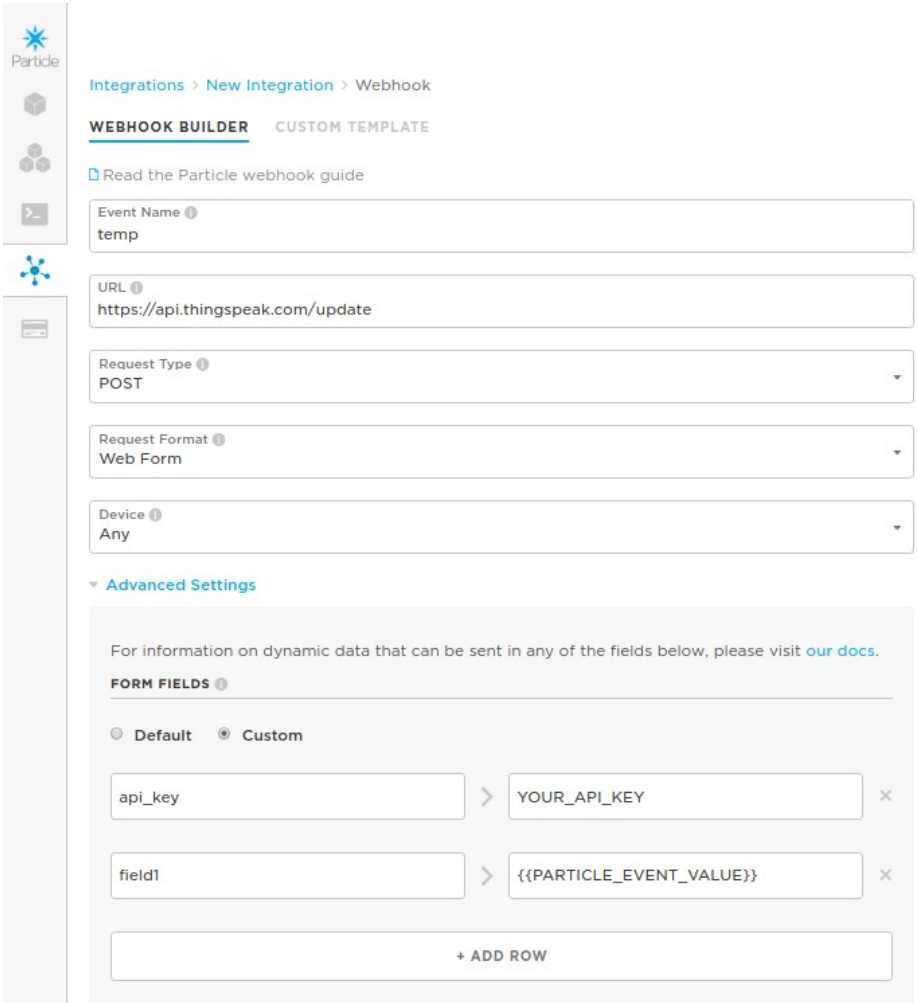
Let's configure our webhook:

- Set the event name to `temp` to match the field in ThingSpeak
- Set the URL to `https://api.thingspeak.com/update`
- Make sure the request type is set to `POST` and the request format is "Web Form"
- If you'd like to limit the webhook triggering to a single one of your devices, choose it from the device dropdown

Next, click on "Advanced Settings," and chose "Custom" in the "Form Fields" section. Drop in the following key/value pairs:

- `api_key` : `YOUR_API_KEY`
- `field1` : `{{PARTICLE_EVENT_VALUE}}`

The form should look something like this:



Click the "Create Webhook" button, and boom! You've created your first webhook.

Taking a step back, we now have a webhook listening for the `temp` event from a Particle device, that will publish temperature data to your ThingSpeak channel. Once the data reaches ThingSpeak, it will be displayed as a line graph.

The last step is getting your Particle device to publish the `temp` event with some temperature information. For this demo, we'll assume that you don't necessarily have a real temperature sensor, so we'll just generate some random data.

Webhook firmware

To get started, go to Particle Build. Create a new app called "TempWebhook."

Copy/paste the following into your application's code:

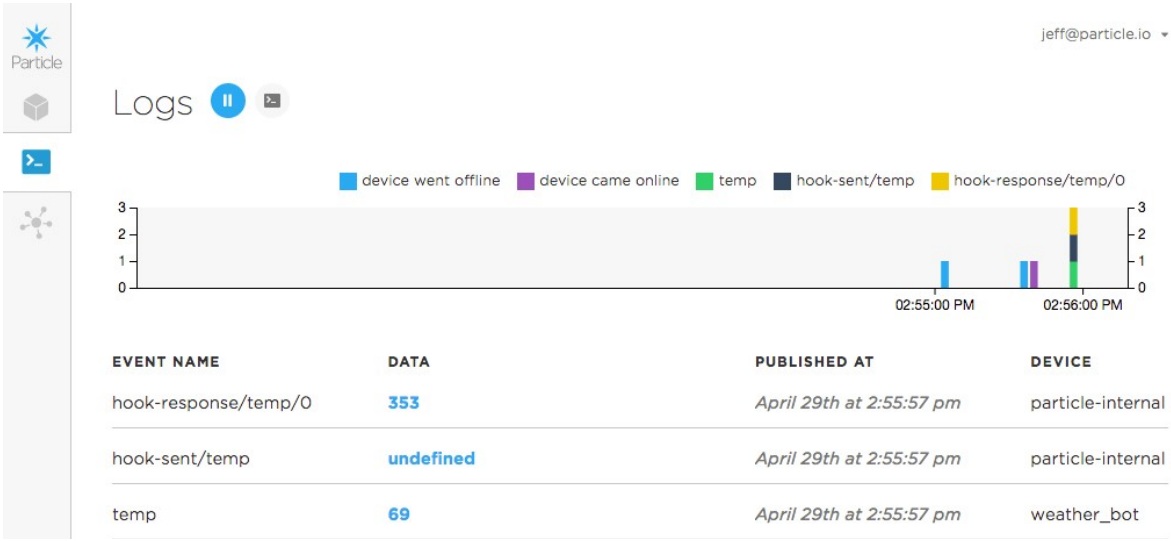
```
int led = D7;  // The on-board LED

void setup() {
  pinMode(led, OUTPUT);
}

void loop() {
  digitalWrite(led, HIGH);   // Turn ON the LED

  String temp = String(random(60, 80));
  Particle.publish("temp", temp, PRIVATE);
  delay(30000);               // Wait for 30 seconds

  digitalWrite(led, LOW);    // Turn OFF the LED
  delay(30000);               // Wait for 30 seconds
}
```

This code will toggle the on-board LED on and off every 30 seconds. When the LED turns on, the device will generate a random temperature value between 60 and 80 degrees and publish it to the cloud with the event `temp`. The temperature value is what will be passed on to ThingSpeak and graphed.

Make sure your device is connected and selected in the IDE (Ensure that the ★ appears next to your device in the Devices pane). Flash the code to your device (Click the ⚡ icon in the sidebar).

Your device should now restart and start publishing the event that will trigger the webhook.

To ensure that everything is working properly, head over to your Logs hub on the console. Every time your webhook triggers, a `hook-sent` event will appear in your user event stream. If the webhook receives a response from the targeted web server with something in the `body`, a `hook-response` event will also appear in your event stream containing the response.



`hook-sent` and `hook-response` events will appear in your event stream for an active webhook

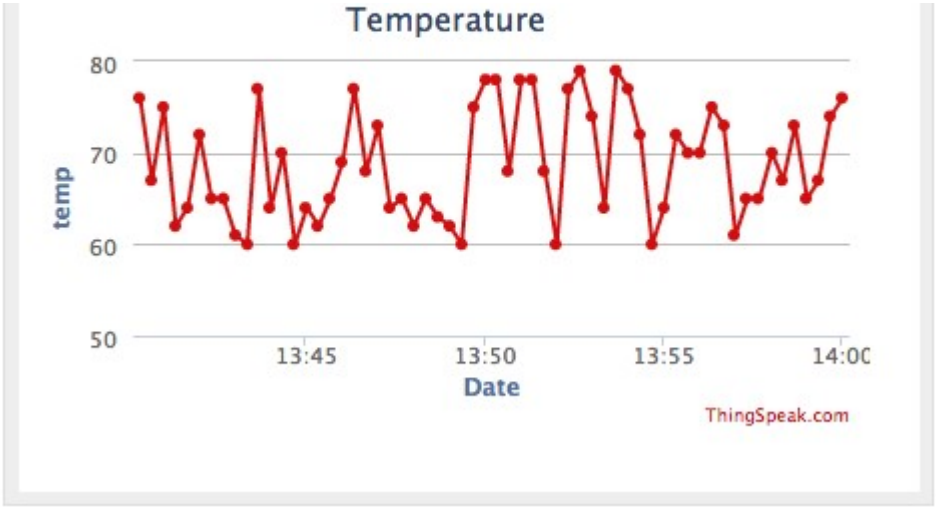
You should see three types of events appearing in your stream:

- `temp` : The original event that your device published that contains the temperature data.
- `hook-sent/temp` : Confirmation that the webhook was sent to ThingSpeak's servers.
- `hook-response/temp/0` : The response that the webhook received from ThingSpeak. According to ThingSpeak's docs, the response is the entry ID of the update. If the update fails, the response is `0` .

Sweet! Things appear to be working properly. Let's check out ThingSpeak and see how our data looks.

See the results

Back on ThingSpeak, navigate back to your channel. You should see temperature data being graphed over time. Your random temperature graph should look something like this:



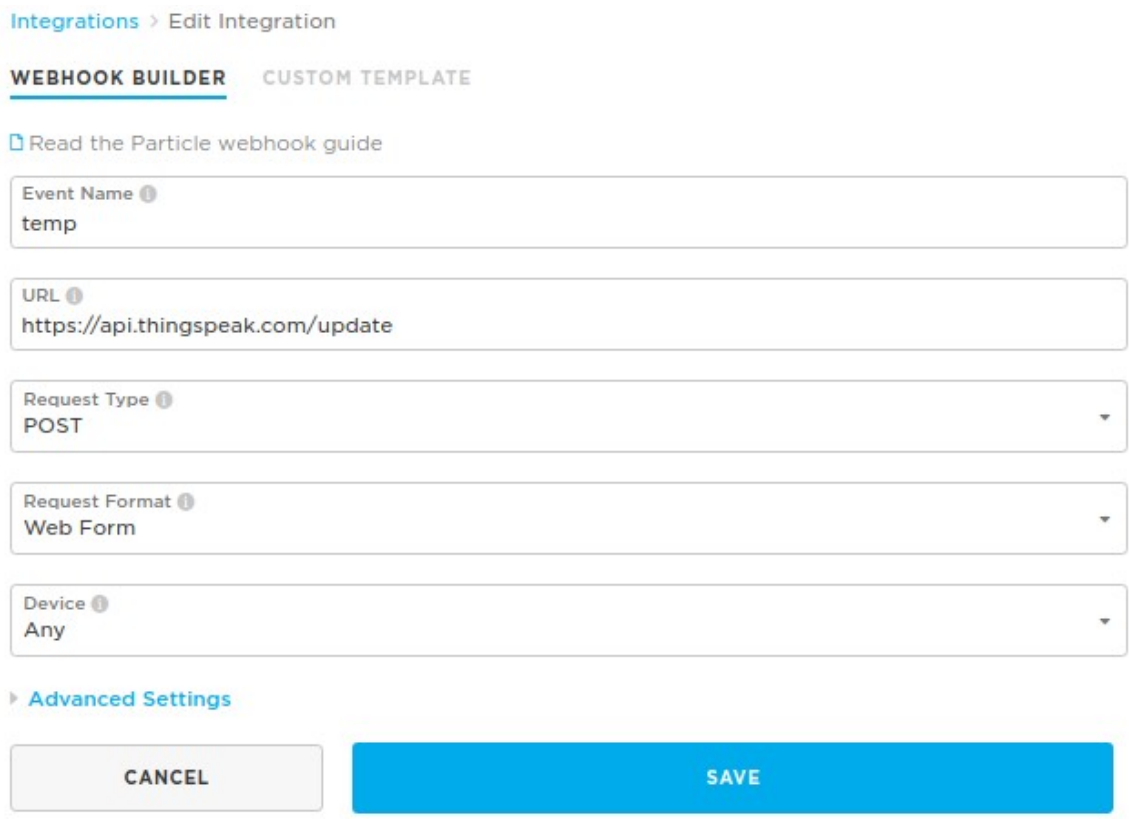
Temperature data from your device being graphed in real-time!

Congratulations! You've created a webhook successfully and gotten data from your connected device into another service. Awesome!

Editing a webhook

To edit a webhook, scroll to the bottom of any webhook page, and press the edit button.

You will be presented with a form, containing the same inputs as the 'Create webhook' page.



If you press 'Cancel', all the changes you made won't be persisted. Clicking on 'Save' updates the integration. When the webhook is fired, it should contain the new information.

Triggering a webhook

In order to signal to the Particle cloud that the webhook should be triggered, your device must publish an event in its firmware. A webhook that has been configured with the event name `temp` would trigger with the following firmware:

```
void loop() {
  // Get some data
  String data = String(10);
  // Trigger the webhook
  Particle.publish("temp", data, PRIVATE);
  // Wait 60 seconds
  delay(60000);
}
```

Breaking this down:

- `Particle.publish()` is the general command for publishing events to the cloud
- `temp` is the name of the event that will trigger the webhook
- `data` is sent along with the event, which will be included with the webhook's HTTP request
- `PRIVATE` ensures that the event will only appear on your private event stream

Getting the response

When using webhooks, it's very common that the targeted web service will return a useful response from the HTTP request containing data that should be sent back to a device. An example of this is triggering a `GET` request to a weather API, and sending the current weather information back to the device that triggered the webhook.

When a web service target by a webhook returns a response with a body, an event is published back to the event stream in the following format:

```
# format for hook response events
hook-response/[triggering-event-name]/[index]
```

Breaking this down:

- All webhook response event names will begin with `hook-response/`,
- Followed by the name of the event that triggered the webhook,
- And finally a numeric index, as the response body is broken into chunks depending on its size before being published to the event stream

You can then subscribe to this event in firmware, if you'd like a device to have access to the webhook response. A snippet of firmware to get a webhook response can look something like this:

```
void setup() {
  // Subscribe to the webhook response event
  Particle.subscribe("hook-response/get_temp", myHandler,
MY_DEVICES);
}

void myHandler(const char *event, const char *data) {
  // Handle the webhook response
}
```

In this example, the event that triggered the webhook, `get_temp`, would result in a webhook response event name of `hook-response/get_temp`. You don't need to worry about the index when subscribing, as the device will receive all events beginning with `hook-response/get_temp`. You'll also notice that you'll need a function that will handle the `hook-response` event. This function will receive the event and its data as arguments.

It is worth mentioning that you can override the default response event name if you'd like. This is useful for product webhooks when you'd like to ensure that only the device that triggered the webhook receives its response. More on that here.

What data gets sent?

When a webhook gets triggered, some data will be sent to the third-party web service by default along with the HTTP request. The default data is:

```
{
    "event": [event-name],
    "data": [event-data],
    "published_at": [timestamp],
    "coreid": [device-id]
}
```

This is same data you'd see if you subscribed to your event stream.

These properties will all be strings except for `published_at`, which is an ISO8601 date formatted string, which tends to be in the form `YYYY-MM-DDTHH:mm:ssZ`.

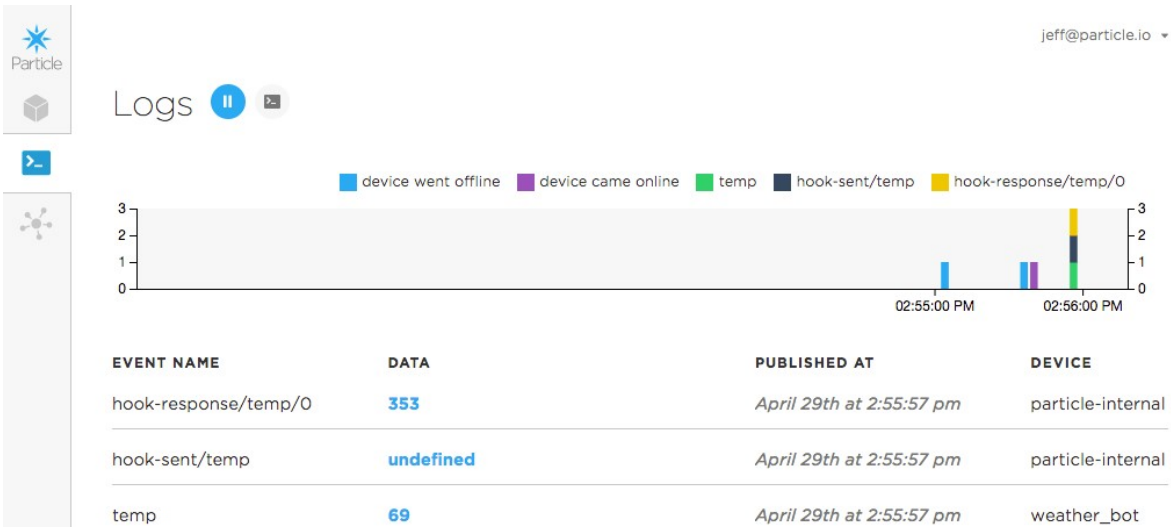
You can customize the format of the data sent with the webhook by changing the "Request Format". When the "Request Type" is `POST`, `PUT` or `DELETE`, the data will be in the request body. You can select "Web Form" (similar to submitting a form from a browser), JSON (common for API requests) or write your own "Custom Body" using the webhook template language. When the "Request Type" is `GET`, the data can only be sent in the "Query Parameters".

You can also customize the structure of the data that gets sent. In the "Advanced Settings" of the Webhook Builder, either keep the "Default" data and add some more fields, or switch to "Custom" and define your own mapping.

Monitoring your webhooks

The easiest way to observe webhook activity is on the Logs hub of your Particle Console. Every time your webhook triggers, a `hook-sent` event will appear in your user event stream. This is confirmation that the Particle cloud successfully forwarded your event to your webhook's target URL.

If the webhook receives a response from the targeted web server with something in the body, a `hook-response` event will also appear in your event stream containing the response. This event will *only* appear in your event stream if the web service returned something in the `body` of its response to the Particle cloud.



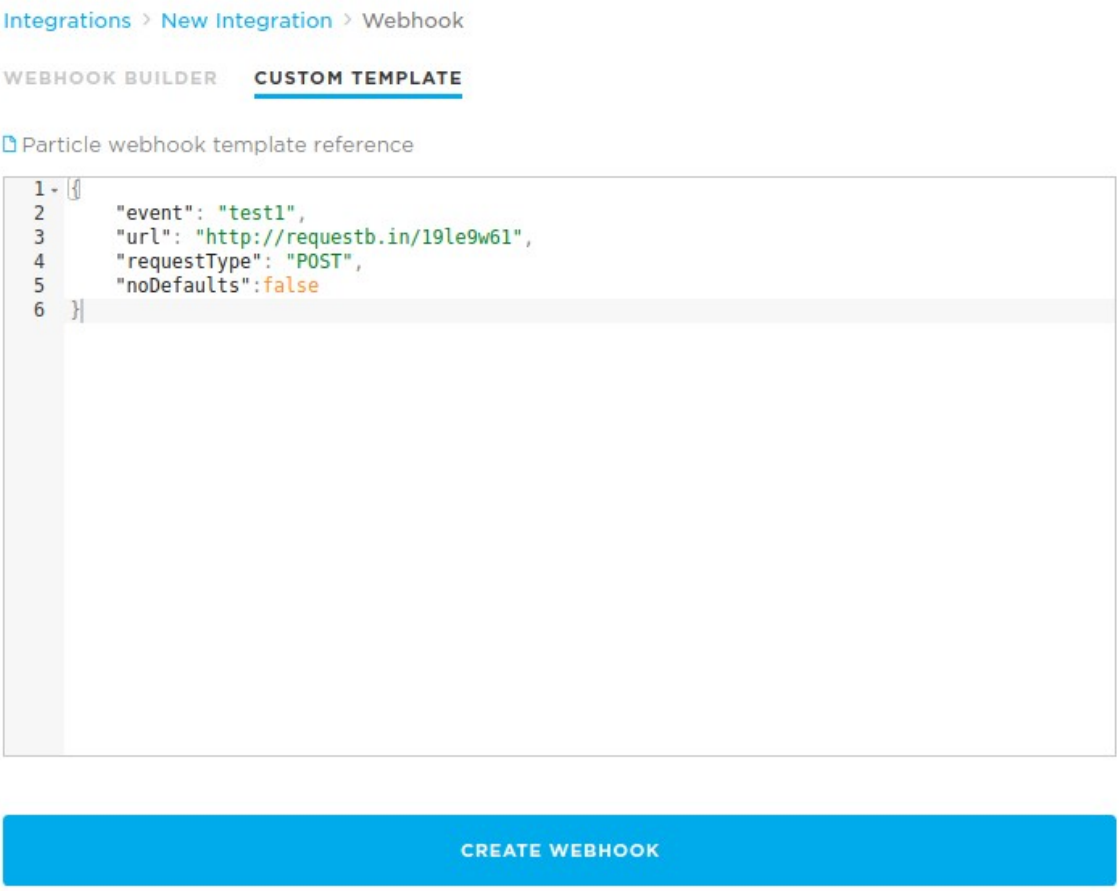
`hook-sent` and `hook-response` events will appear in your event stream for an active webhook

It is also possible that you can see errors appear in your Logs from unsuccessful attempts to contact the third-party server. You can read more about those here.

Note: This method of monitoring activity is not enabled for product-level webhooks. A method for monitoring product-level webhooks is coming soon.

Custom Template

The "Custom Template" tab of the webhook editor shows the raw configuration for the webhook. The syntax is described in the webhook reference page.

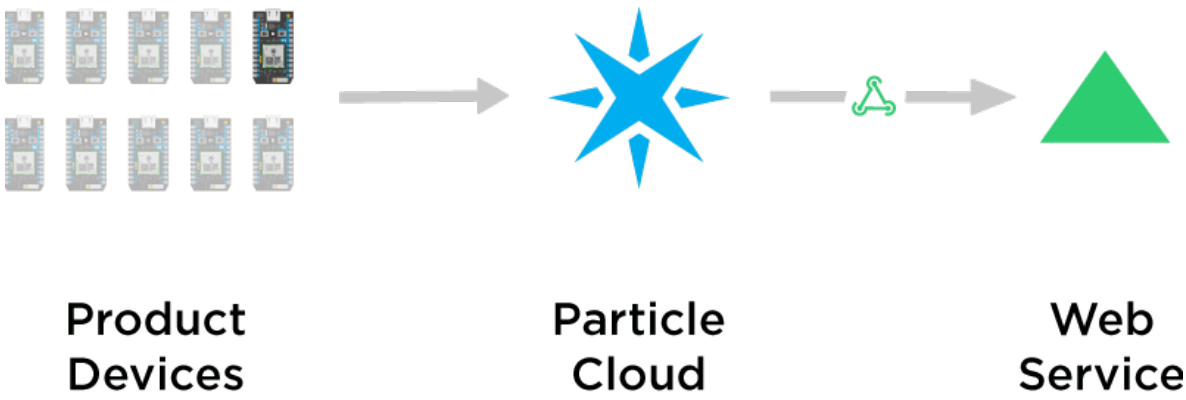


If you want to create a webhook from an existing template, you can switch over to the "Custom Template" tab of the webhook editor and paste in a JSON webhook template. You can even switch back to the "Webhook Builder" and continue making some edits.

You can also copy from the "Custom Template" tab and share the webhook template with others.

Product webhooks

If you are building a product using Particle, you now have the ability to create webhooks at the product-level. This will allow you as a product creator to define a single webhook than any of the devices in the product's fleet can trigger.



Create a single webhook that any of your product devices can trigger

As devices in your product's fleet will be running the same firmware, product webhooks are a scalable way to integrate with third-party web services. Trigger a product webhook when you'd like to do thing like:

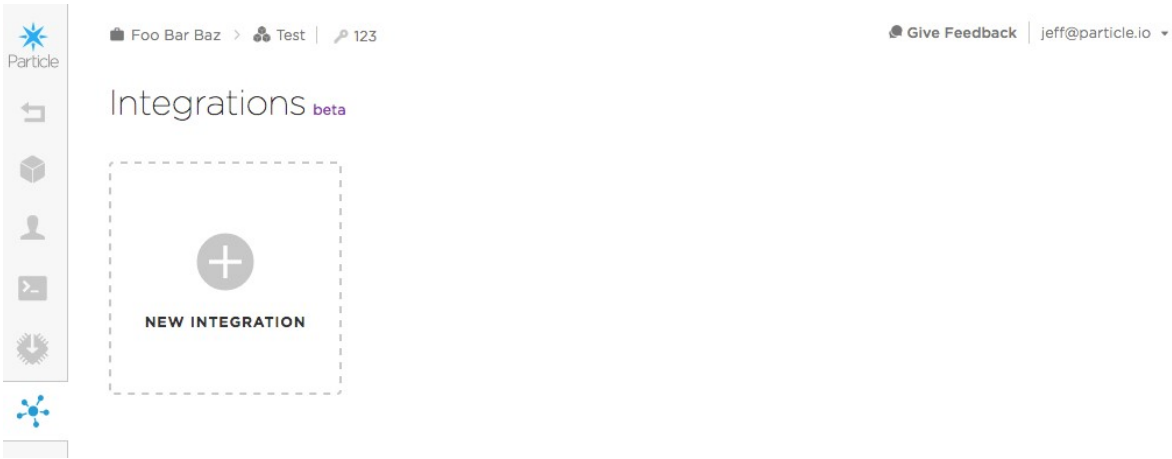
- Sending information about how a customer's device is behaving to an analytics service
- Make an API call to your servers to send personalized content to a device

- Save data to a hosted database in the cloud

Create a product webhook

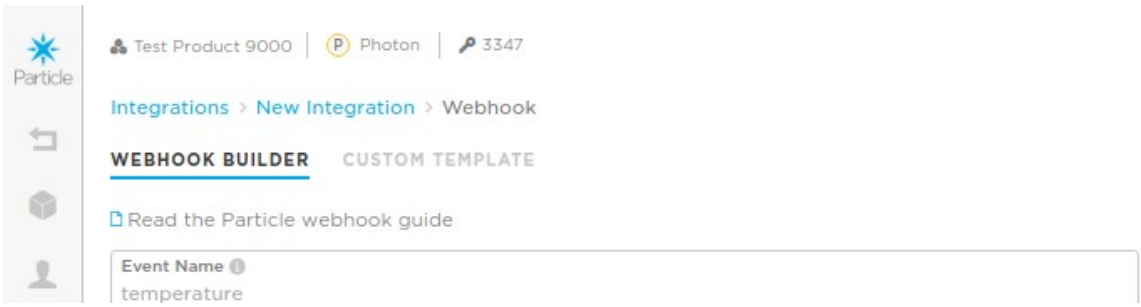
If you don't have one already, you'll need to define a product before you will be able to create product webhooks. Currently, webhooks for products are in beta and will evolve over the coming months.

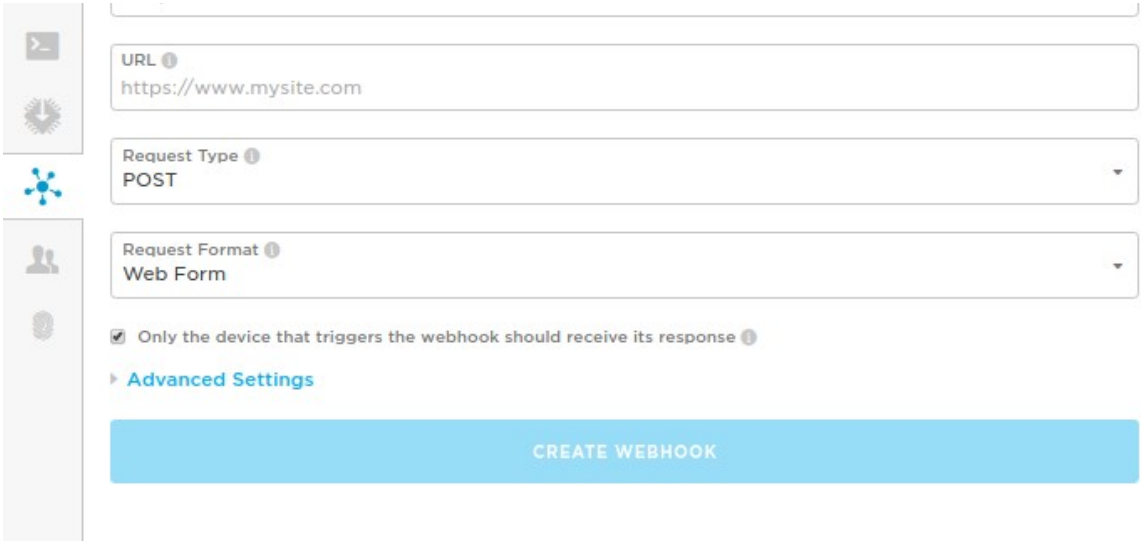
Product webhook management can also be done from the Particle Console. To create a webhook, navigate to your product's hub on the console and click on the Integrations tab (). You'll see a very similar view to the Integrations hub as a developer.



Product integrations are currently in public beta

Click on "New Integration" -> "Webhook." Again, the view will be very similar to what you would see in your developer console. However, you will notice that the "Devices" dropdown has been replaced by a checkbox. This has to do with responses to product webhooks, which we'll cover in the next section.





Create a product webhook from the Particle Console

Product webhook responses

For product webhooks, because *any* device in the fleet can trigger the webhook, how can we ensure that *only* the device that triggers the webhook will receive its response? After all, we wouldn't want a device in Phoenix receiving weather data that another device asked for in Chicago.

This is where the checkbox discussed in the previous section comes into play. The one that says: "Only the device that triggers the webhook should receive its response." This setting (enabled by default) will individualize the webhook response so that it can be routed correctly to the triggering device.

As discussed earlier, any response from a webhook will result in a `hook-response/[event_name]` event in the event stream. Normally, if you wanted to get that response on your device, you would add something like this to your firmware: `Particle.subscribe("hook-response/weather/", myHandler, MY_DEVICES);`

If you used this line of code in product firmware, however, a given device listening for a webhook response could receive it from *any device in the fleet*, not just the webhook that it triggered.

Ensuring that the "Only the device that triggers the webhook should receive its response" checkbox is checked will prepend the device ID of the triggering

webhook to the `hook-response` event. This will allow you to write firmware that will listen to only webhook responses for that particular device, like this:

```
void setup() {
  // Subscribe to the response event, scoped to webhooks triggered by this device
  Particle.subscribe(System.deviceID() + "/hook-response/weather/", myHandler, MY_DEVICES);
}

void myHandler(const char *event, const char *data) {
  // Handle the webhook response
}
```

At any time, you can see some sample firmware for both triggering and getting responses from webhooks on your Particle Console. To do this, simply click on one of your product webhooks and scroll down to "Example Device Firmware."

Monitoring Product Webhooks

[Coming Soon]

Debugging with RequestBin

Depending on the service you're sending data to, it can be difficult to debug a webhook, especially if you're sending data using templates. A great debugging tool is the free service http://requestb.in/. You create a new RequestBin and it returns a URL that you use as the URL in your webhook. Then, when you refresh you RequestBin page, it will show you the requests that have come in, with all of the parameters and data. Very handy!

Here's a simple webhook template. Save it in a file "hook1.json".

```
{
    "event": "test1",
    "url": "http://requestb.in/19le9w61",
    "requestType": "POST",
```

```
    "noDefaults":false
}
```

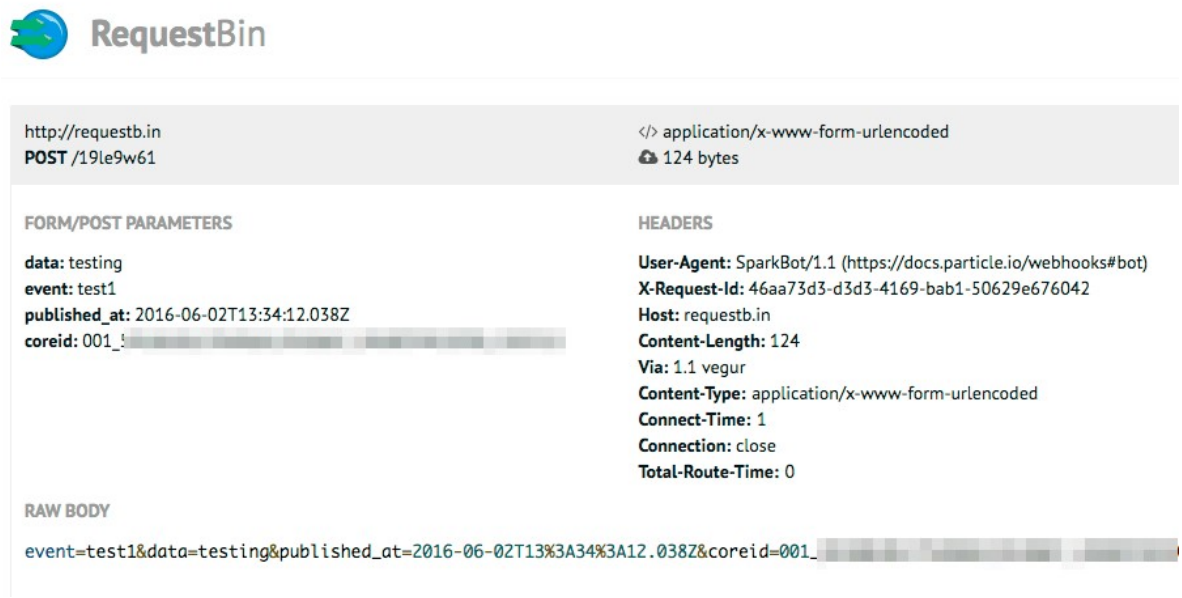
You can create a webhook through the Console by using the "Custom Template" tab of the new webhook form or through the Particle CLI by issuing the command:

```
particle webhook create hook1.json
```

Tested it first using the CLI:

```
particle publish test1 "testing" --private
```

And this is what the request bin looks like:



In this simple example, you can see the POST request with the default data in form encoding.

Here's an example using JSON encoding.

```
{
    "event": "test1",
```

```
    "url": "http://requestb.in/19le9w61",
    "requestType": "POST",
    "json": {
        "name": "",
        "value": ""
    },
    "noDefaults": true
}
```

Generating the event:

```
particle publish test1 "testing2" --private
```

And the RequestBin results:

Advanced Topics

See the webhook reference for more details on customizing webhooks with variables, examples of different webhook configurations as well as community guides on setting up webhooks with external services.

As a quick reference, these are the pre-defined webhook variables available for you to use:

- `{{PARTICLE_DEVICE_ID}}` : The ID of the device that triggered the webhook
- `{{PARTICLE_EVENT_NAME}}` : The name of the event that triggers the webhook
- `{{PARTICLE_EVENT_VALUE}}` : The data associated with the webhook event
- `{{PARTICLE_PUBLISHED_AT}}` : When the webhook was sent

Also, check out and join our community forums for advanced help, tutorials, and troubleshooting.


THE INTERNET BUTTON

Congratulations on receiving your brand new Internet Button! This part of the guide will help you start working with your Internet Button so that you can connect to the Internet in an entirely new way.

Unboxing

Your Internet Button comes in a .

Inside the tin, you will find:

- (1)
- (1) to prevent damage to your Photon
- (1) to help diffuse the light from the LEDs
- (1) resting between the two removeable covers

If you , you can see some of the components. There are four buttons, a buzzer, and a space for a JST connector if you prefer battery power to USB power.

You can also remove the module cover and Photon, then the shield cover, to expose the . Here, you can see the exposed LEDs and and the accelerometer.

For more information, check out the Internet Button datasheet.

Connecting

Plug in your Photon and go through the setup process as seen in either the start (smartphone connection) or connect (USB serial connection) sections.

Examples

You can find examples on how to use your Internet Button in the official Internet Button library under the Libraries tab on Particle Build. You can also access them via this GitHub repo, or by checking out the text below.

You should be able to fork the examples from Particle Build or copy and paste the code from the GitHub repo or from this page. We recommend going through these examples in order for the best understanding of how the Internet Button works. If you haven't used Particle Build before, read the Particle Build guide.

If you copy and paste these examples, make sure that you include the official Internet Button library before flashing the code. Instructions on how to include a library can be found here.

Blink an LED

```
#include "InternetButton/InternetButton.h"

// Create a Button named b. It will be your friend, and you two
will spend lots of time together.
```

```
// You may be wondering about those two slashes and this gray text- they're called comments, and
// don't affect the code. Think of this as the voice of the narrator.
InternetButton b = InternetButton();

// The code in setup() runs once when the device is powered on or reset. Used for setting up states, modes, etc
void setup() {
    // Tell b to get everything ready to go
    // Use b.begin(1); if you have the original SparkButton, which does not have a buzzer or a plastic enclosure
    // to use, just add a '1' between the parentheses in the code below.
    b.begin();
}

/* loop(), in contrast to setup(), runs all the time. Over and over again.
Remember this particularly if there are things you DON'T want to run a lot. Like Particle.publish() */
void loop() {
    // Let's turn an LED on. How about #6, which is at the 6 o'clock position? Let's make it blue and bright.
    b.ledOn(6, 0, 0, 255);
    // The format here is (LED, red, green, blue), so we're making a color with no red or green, but ALL the blue
    // You should know that the range of brightness here is 0-255, so 0 is off and 255 is the most possible.
    // After you use this code, try making the LED white- all the red, green, and blue.

    // Since the LED is now on, let's have it stay that way for one second
    // Delay pauses the code for the amount of time given, in milliseconds- so 1000 millis is one whole second
    delay(1000);

    // And to blink the LED, we'll need to turn it back off and then pause for another second
    b.ledOff(6);
    delay(1000);

    // Now you're blinking! Play with which LED is blinking
```

```
(1-11), the delays between, and the color.
}
```

Blink All LEDs

```
#include "InternetButton/InternetButton.h"

/* This SparkButton library has some useful functions.
Here we blink ALL the LEDs instead of just one.*/

InternetButton b = InternetButton();

void setup() {
    // Tell b to get everything ready to go
    // Use b.begin(1); if you have the original SparkButton, which
does not have a buzzer or a plastic enclosure
    // to use, just add a '1' between the parentheses in the code
below.
    b.begin();
}

void loop(){
    b.allLedsOn(0,20,20);
    delay(1000);
    b.allLedsOff();
    delay(1000);

    // Notice that I made them much dimmer, so it's a bit less
painful
}
```

Buttons and LEDs

```
#include "InternetButton/InternetButton.h"

/* How about we make this interactive? */

InternetButton b = InternetButton();
```

```
void setup() {
    // Tell b to get everything ready to go
    // Use b.begin(1); if you have the original SparkButton, which
does not have a buzzer or a plastic enclosure
    // to use, just add a '1' between the parentheses in the code
below.
    b.begin();
}

void loop(){

    // When you click the second button (at the 3 o'clock
position) let's turn that LED on
    if(b.buttonOn(2)){
        b.ledOn(3, 255, 255, 255);
    }
    // And if the button's not on, then the LED should be off
    else {
        b.ledOff(3);
    }

    /* Much like the LEDs, there are also functions to check if
all the buttons are on- b.allButtonsOn()
    or if all the buttons are off- b.allButtonsOff() */
}
```

Button/Light Combo

```
#include "InternetButton/InternetButton.h"

/* Here's a nice combination of features that I like to use.
Note the use of the allButtons function. */

InternetButton b = InternetButton();

bool rainbow_mode = false;

void setup() {
    // Tell b to get everything ready to go
    // Use b.begin(1); if you have the original SparkButton, which
does not have a buzzer or a plastic enclosure
    // to use, just add a '1' between the parentheses in the code
```

```
above.
    b.begin();
}

void loop(){

    // If this calls for a full spectrum situation, let's go
rainbow!
    if(b.allButtonsOn()) {
        // Publish the event "allbuttons" for other services like
IFTTT to use
        Particle.publish("allbuttons",NULL, 60, PRIVATE);
        b.rainbow(5);
        rainbow_mode = true;

        // If all buttons are on, don't try to process
        // the individual button responses below.  Just return.
        return;
    }

    // If we are not in rainbow mode anymore, turn the LEDs off
    if (rainbow_mode == true) {
        b.allLedsOff();
        rainbow_mode = false;
    }

    // Process individual buttons and LED response
    if (b.buttonOn(1)) {
        b.ledOn(12, 255, 0, 0); // Red
        // Publish the event "button1" for other services like
IFTTT to use
        Particle.publish("button1",NULL, 60, PRIVATE);
        delay(500);
    }
    else {
        b.ledOn(12, 0, 0, 0);
    }

    if (b.buttonOn(2)) {
        b.ledOn(3, 0, 255, 0); // Green
        // Publish the event "button2" for other services like
IFTTT to use
        Particle.publish("button2",NULL, 60, PRIVATE);
        delay(500);
    }
```

```
    else {
        b.ledOn(3, 0, 0, 0);
    }

    if (b.buttonOn(3)) {
        b.ledOn(6, 0, 0, 255); // Blue
        // Publish the event "button3" for other services like
IFTTT to use
        Particle.publish("button3",NULL, 60, PRIVATE);
        delay(500);
    }
    else {
        b.ledOn(6, 0, 0, 0);
    }

    if (b.buttonOn(4)) {
        b.ledOn(9, 255, 0, 255); // Magenta
        // Publish the event "button4" for other services like
IFTTT to use
        Particle.publish("button4",NULL, 60, PRIVATE);
        delay(500);
    }
    else {
        b.ledOn(9, 0, 0, 0);
    }

    if(b.allButtonsOff()) {
        // Do something here when all buttons are off
    }
}
```

Using The Accelerometer

```
#include "InternetButton/InternetButton.h"
#include "math.h"

/*Did you know that the SparkButton can detect if it's moving?
It's true!
Specifically it can read when it's being accelerated. Recall that
gravity
is a constant acceleration and this becomes very useful- you know
the orientation!*/
```

```
InternetButton b = InternetButton();

void setup() {
    //Tell b to get everything ready to go
    // Use b.begin(1); if you have the original SparkButton, which
does not have a buzzer or a plastic enclosure
    // to use, just add a '1' between the parentheses in the code
below.
    b.begin();
}

void loop(){

    //How much are you moving in the x direction? (look at the
white text on the board)
    int xValue = b.readX();

    //How about in the y direction?
    int yValue = b.readY();

    //And the z!
    int zValue = b.readZ();

    //This will make the color of the Button change with what
direction you shake it
    //The abs() part takes the absolute value, because negatives
don't work well
    b.allLedsOn(abs(xValue), abs(yValue), abs(zValue));

    //Wait a mo'
    delay(50);
}
```

Orientation Awareness

```
#include "InternetButton.h"
#include "math.h"

/* Did you know that the Internet Button can detect if it's
moving? It's true!
```

```
Specifically it can read when it's being accelerated. Recall that
gravity
is a constant acceleration and this becomes very useful- you know
the orientation!*/

InternetButton b = InternetButton();
int ledPos = 0;

void setup() {
    // Tell b to get everything ready to go
    // Use b.begin(1); if you have the original SparkButton, which
does not have a buzzer or a plastic enclosure
    // to use, just add a '1' between the parentheses in the code
below.
    b.begin();

    // reduce to less than full eye-blazing brightness
    b.setBrightness(95);

    Particle.variable("ledPos",ledPos);
}

void loop(){
    // previous LED off (or 'null' LED0 off the first time through)
    b.ledOn(ledPos, 0, 0, 0);

    // Want to figure out which LED is the lowest?
    // We've hidden the necessary trigonometry in this function.
    ledPos = b.lowestLed();

    // give some time for human retinal response
    delay(330);

    // Now turn the lowest LED on
    b.ledOn(ledPos, 0, 30, 30);

    // Wait a mo'
    delay(330);
}
```

Connecting to the Internet

```
#include "InternetButton/InternetButton.h"
#include "math.h"

/* Let me show you how easy it is to put the Button on the
Internet.
Useful info, like how to access the data from your browser, can be
found here: https://docs.particle.io/photon/firmware/#particle-
function-
The code to control the number of illuminated LEDs is here:
https://github.com/particle-
iot/InternetButton/blob/master/controlKnob.html
Try naming one of your devices "InternetButton" and running
controlKnob in your browser or on your phone!
Note that the Core or Photon *must* be named "InternetButton"
because the javascript looks for it.
*/

InternetButton b = InternetButton();
float brightness = 0.1;
int red, green, blue;
int howMany = 6;
int whichColor = 100;
bool changed = false;

void setup() {
    // Use b.begin(1); if you have the original SparkButton, which
does not have a buzzer or a plastic enclosure
    // to use, just add a '1' between the parentheses in the code
below.
    b.begin();

    //This is all you need to make the function controller()
available to the internet
    //The API name and the local name don't need to be the same;
just my style
    Particle.function("controller", controller);

    //This function figures out what combination color, brightness
and LEDs to display
    makeColors();
```

```
}

void loop(){
    //Clicking "up" makes the LEDs brighter
    if(b.buttonOn(1)){
        if(brightness < 1){
            brightness += 0.005;
            changed = true;
        }
    }
    //Clicking "down" makes the LEDs dimmer
    else if (b.buttonOn(3)){
        if(brightness > 0){
            brightness -= 0.005;
            if(brightness < 0){
                brightness = 0;
            }
            changed = true;
        }
    }
    //Clicking "right" and "left" change the color
    else if (b.buttonOn(2)){
        if(whichColor < 255){
            whichColor += 1;
            changed = true;
        }
    }
    else if (b.buttonOn(4)){
        if(whichColor > 0){
            whichColor -= 1;
            changed = true;
        }
    }

    //If anything's been altered by clicking or the
Particle.function, update the LEDs
    if(changed){
        delay(10);
        makeColors();
        changed = false;
    }
}

//Uses the brightness and the color values to compute what to show
```

```
void makeColors(){
  uint8_t x = whichColor;
  if(x < 85) {
    red = brightness * x * 3;
    green = brightness * (255 - x * 3);
    blue = 0;
  } else if(x < 170) {
     x -= 85;
     red = brightness * (255 - x * 3);
     green = 0;
     blue = brightness * x * 3;
  } else {
     x -= 170;
     red = 0;
     green = brightness * x * 3;
     blue = brightness * (255 - x * 3);
  }

  b.allLedsOff();
  for(int i = 1; i <= howMany; i++){
    b.ledOn(i, red, green, blue);
  }
}

/*
controller() is the local function that is executed when the API
function "controller" is called.
It changes how many LEDs on the Button are illuminated.
*/
int controller(String command){
    //parse the string into an integer
    int state = atoi(command.c_str());

    //Check that the value it's been given is in the right range
    if (state > 11) {state = 11;}
    else if (state < 0) {state = 0;}

    howMany = state;

    changed = true;

    return 1;
}
```


FIRMWARE LIBRARIES

Overview

Libraries are a central part of project development on the Particle platform leveraged by thousands of engineers in our community. Simply stated, a Particle library is *a collection of reusable firmware code that can be easily added to one or many Particle projects.*

Leveraging high quality libraries to build Internet-connected projects and applications can significantly reduce the risk, time, and cost associated with creating IoT product or device. Libraries also make it possible to easily maintain and reuse common code blocks, as well as incorporate and leverage existing third party libraries from the Particle ecosystem.

In general, libraries in the Particle ecosystem have the following features:

1. **Most Arduino libraries are compatible with Particle.** We've worked hard to ensure that our firmware API contains all of the most commonly used Arduino functions and firmware commands so that many Arduino libraries can be submitted into the Particle library ecosystem without modification. All of the most popular Arduino libraries are already available through our libraries system, and many others can be easily modified for compatibility.
2. **Particle libraries can include and depend on other Particle libraries.** If your library requires another external library as a dependency, it is easy to specify the particular library and even version of the library that your library depends on. A good example is our `internet-button` library, which depends on the popular `neopixel` library for controlling NeoPixel LEDs. You can learn more about libraries with dependencies in the Library file structure section below.
3. **Particle libraries are reliable.** In addition to building and sharing our own high quality libraries, Particle verifies and promotes high quality community libraries that are fully documented, perform reliably, and include a variety of usage examples. Using our official and verified libraries means you'll spend less time debugging and more time building your project.

Kinds of libraries

Public Libraries

The vast majority of Particle libraries are developed and maintained by the Particle community and made available for broader use via the Particle libraries ecosystem. All public libraries are available for public consumption through our development tools and via our Libraries API. The availability of such a large number of libraries in a single place makes developing IoT products on the Particle platform fast and simple.

Note that a library may have its own associated open source license that limits or restricts redistribution or commercialization of the library.

Official Libraries

Official libraries are libraries that were created *by members of the Particle team* and are designed to be used with Particle hardware. Examples of Official Particle libraries include:

- `internetbutton` for the Internet Button, our kit for quickly prototyping simple IoT projects and experiences
- `makerkit` for our Maker Kit, our kit for beginners to learn how to build IoT projects
- `relayshield` for the Relay Shield, our shield for switching high and low voltage electronics.
- `assettracker` for the Asset Tracker Shield, our kit for tracking and locating valuable possessions

All Particle libraries meet the same quality standards as Verified libraries, and appear in the library list with the Particle logo next to them.

Verified Libraries

Verified libraries are community-contributed libraries that have been reviewed and confirmed by members of the Particle team to meet the following criteria:

1. **The library is well documented.** It contains in-line comments that document proper usage of the library and contains example applications that demonstrate how to use each of the included functions.
2. **The library has been reviewed for quality.** The library compiles on all relevant hardware platforms and performs as intended. The library includes testing instructions (`verification.txt`) that anyone can follow to independently verify that the library is working as expected.
3. **The library has improved visibility.** Verified libraries float to the top of library searches, improving the visibility of the library within the Particle ecosystem.

Private Libraries

Private libraries are libraries that have been uploaded to the Particle Cloud for reuse with many projects, but are *only* visible to the individual who created and submitted the library. Private libraries can be published as public libraries at any time by the author of the library using the `particle library publish` command.

Library file structure

Overview

Every Particle library complies with the following file structure that will be automatically generated when a new library is initialized:

- `examples`
    - `usage`
        - `usage.ino`
- `src`

    - `mylibrary.cpp`
    - `mylibrary.h`
- `library.properties`
- `README.md`
- `LICENSE.txt`

`examples` is the folder that contains the example applications that reference your library with one example per directory. If your library controls LEDs, for example, you should include an example called `examples/control/control.ino` that demonstrates how someone could use the library in a typical application.

`src` is the folder that contains the actual library files (`.cpp` and `.h` files) that define the library's behavior. Everything in the `src` folder will be compiled when a user adds the library to their project and compiles it. You can add subfolders to `src` if you have many files in your library.

`library.properties` includes descriptive information about your library (title, description, version, author, license), and also specifies any other libraries that your library depends on. Libraries dependencies can be tagged at a particular version in this file.

`README.md` provides instructions for library creators on creation and usage.

`LICENSE.txt` is the file that defines the license that the public library is distributed with. All libraries in Particle's library ecosystem must include an associated license.

library.properties fields

- **name** A short name for the library. The name must be unique, so there aren't 2 libraries with the same name. It will be the name of main `.cpp` and `.h` file.
- **version** A Semantic Versioning version number like 1.0.0
- **author** Your name and email like `The Author <author@example.com>`. If there are several authors, separate them with commas.
- **license** The acronym for the license this library is released under, such as GPL, MIT, Apache-2.

- **sentence** A one sentence description of the library that will be shown in the library browser.
- **paragraph** A longer description to be shown in the details page of the library. This is always shown after **sentence** so start with the second sentence of the description.
- **url** The web page that a user wanting more information would visit like a GitHub URL.
- **repository** The git repository like `http://github.com/user/project.git`, if available.
- **architectures** A comma-separated list of supported hardware boards that are compatible with your library. If missing or set to `*`, the library will be available for all architectures. Available values for Particle libraries: `spark-core`, `particle-photon`, `particle-electron`, `particle-p1`, `digistump-oak`, `bluz`, `redbear-duo`.
- **dependencies.** Other library that this library depends on, one line per library dependency. The value is the desired version number of the library dependency.
- **whitelist** Additional files to include when publishing a library. By default these files are included in a library when publishing: `*.ino`, `*.pde`, `*.cpp`, `*.c`, `*.c++`, `*.h`, `*.h++`, `*.hpp`, `*.ipp`, `*.properties`, `*.md`, `*.txt`, `*.S`, `*.a` and `LICENSE`.

Project file structure

There are 3 kinds of project structure:

- legacy
- simple
- extended

Legacy Structure

The legacy project structure stores files in the root of the project. There is no project definition file. This is the structure used by all projects prior to libraries v2.

- `myapp`
    - `application.ino`

A legacy project does not support using libraries.

Simple Structure

The simple project structure is similar to the legacy structure - the project sources are stored in the root. However, the project also includes a project definition file `project.properties`. Even saving a blank `project.properties` file is enough to make a simple project.

- `myapp`
    - `application.ino`
    - `project.properties`

A simple project has standard support for libraries; libraries can be added to the project via the CLI `library add` command or the Desktop IDE library manager.

Extended Structure

The extended structure expands on the simple structure, placing all application sources in the `src` folder.

- `myapp`
    - `project.properties`
    - `src`
        - `application.ino`

An extended project has full support for libraries, supporting both `library add` and copied libraries.

An extended project can be created by using the Desktop IDE "Start a new project" or the CLI `particle project create` command.

Using libraries

Libraries consumption is supported in each of our three primary development tools. Instructions for using libraries can be found in the documentation for each of those tools, linked below:

- Using libraries with the Web IDE
- Using libraries with the Desktop IDE
- Using libraries with the Command Line Interface (CLI)

Contributing libraries

The main steps for contributing a library are preparing the structure, writing the code, testing the examples, uploading a private version and publishing to the public.

Library contribution is currently supported in our Desktop IDE and via our Command Line Interface (CLI).

- Contributing libraries with the Desktop IDE - *Coming soon!*
- Contributing libraries with the Command Line Interface (CLI)

Preparing the structure

There are several ways to start contributing a Particle library.

You can start a brand new library by generating the file structure with the Command Line Interface (CLI) using the `particle library create` or downloading the example library and editing it.

If you made a library in the past, you can migrate it to the new format with `particle library migrate`.

You can modify an existing library by forking the code from GitHub. You can also download an existing library through the CLI with `particle library view`

`<library_name>`. Make sure you move the library to a new folder before modifying it in that case.

You can also start with an existing Arduino library from GitHub. Particle libraries have the same structure as Arduino libraries.

Writing the code

The main sources of the library go into `src/lib_name.cpp` and `src/lib_name.h`. More complex libraries may use a nested structure within the `src/` directory. For example `src/subFolder/subhHeader.h` and referenced as `#include <subFolder/subHeader.h>`.

Create at least one example `.ino` file inside a subfolder of `examples` to show people how to use the library.

If your library depends on other libraries you can add those dependencies to `library.properties` with `particle library add`. For example, since the Internet Button contains NeoPixel LEDs, the `InternetButton` library has the line `dependencies.neopixel=0.0.10` in `library.properties` to indicate this.

List the hardware platforms supported by your library supports to the `architectures` field in `library.properties`. In the code you can compare the current platform constant with the platform IDs.

```
#if PLATFORM_ID == 10 // Electron
    #include "cellular_hal.h"
#elif PLATFORM_ID == 6 // Photon
    #include "softap_http.h"
#else
    #error "This library only works on the Electron and Photon"
#endif
```

Review the `library.properties` and `README.md` and fill in as much information as possible.

Testing the examples

The best way to ensure people will use your library is to provide good examples.

Once you've written the first draft of the library you can test the examples on hardware you own in the CLI with `particle flash <my_device> examples/<example_name>` from the library folder or in the Desktop IDE by right-clicking on the example and selecting "Flash this example".

Uploading a private version

Once your library is ready to go, you can upload it to Particle in the CLI with `particle library upload` from the library folder or in the Desktop IDE with "Upload library".

When you upload a new private version you can use the library as usual from your own account in the Web IDE, Desktop IDE or CLI.

If you find issues, you can upload another private version with the same version number.

When you want to modify an existing library for your own projects only, you can stop after uploading a private version. However if you want other people to be able to use the library go on to publishing it.

Publish to the public

When a version is ready for prime time, simply type `particle library publish <my_lib>` in the CLI and it will make the latest private version available to the public. You can also publish through the Desktop IDE.

After this, anybody with a Particle account will be able to use your library! Thank you!

Migrating Libraries

On January 23, 2017, Particle introduced a new version of our firmware library manager, requiring that libraries be migrated from the old library structure (v1) to our new library structure (v2).

With our original firmware library manager, libraries could only be contributed and consumed through our Web IDE (Build). We've now upgraded the library manager behind the Web IDE, and made those libraries accessible in our Desktop IDE (Dev) and CLI.

Libraries under the new library format have the following features:

- Every library now has a library.properties file that can be used to specify external library dependencies, library version number, description, and associated open-source license
- Libraries are now accessible via our firmware libraries API
- Libraries can now be added to projects via our Desktop IDE and CLI

All existing Particle applications that included a v1 library have been preserved and will continue to function as before. However, all new library includes will pull from our migrated v2 library list, so all new Particle projects that include a library will use the updated library structure.

For that reason, it may be necessary to migrate a library to the new library structure if the library was originally created as a v1 library.

Instructions for migrating v1 libraries to the new library format using the CLI and Desktop IDE are included below.

Using the CLI

Follow these steps to migrate a v1 Particle library to the new v2 structure using the Particle CLI:

- Install the Particle CLI version 1.19 or later.
    - If you do not have the Particle CLI installed on your machine, you can download and install it using our OS-specific instructions, here
    - If you already have the Particle CLI installed, you can update it to the latest version by running `npm update -g particle-cli`
- Run `particle library migrate` in your library directory
- Edit the newly created `library.properties` file to add a GitHub URL to the `url` field (like https://github.com/particle-iot/internetbutton) and the git

remote to the `repository` field (like https://github.com/particle-iot/internetbutton.git)

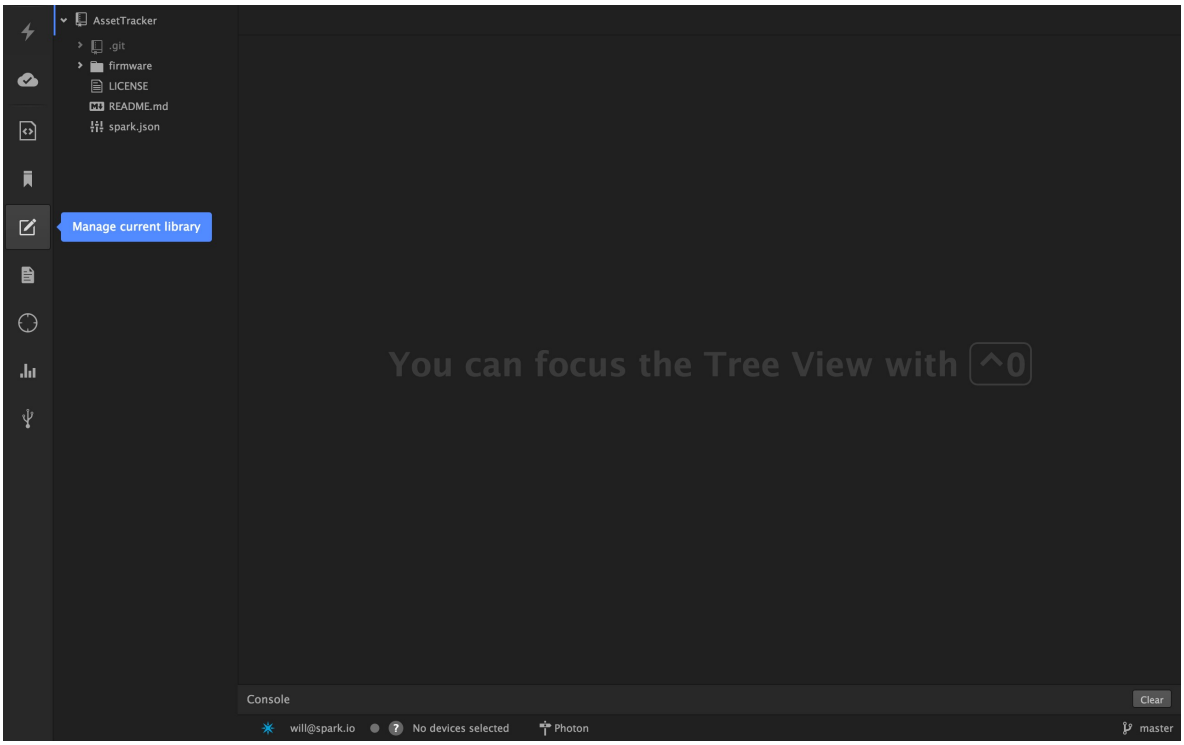
- If your library depends on another library, run `particle library add dependency` in your library directory and remove the source files of the other library from your own repository
- Ensure that the example applications for your library compile by running `particle compile photon examples/<name>` in your library directory
- Refresh the `README.md` file for your library with detailed information and instructions for using and interacting with the library. The `README.md` file will be used as the "home page" for your library.
    - See https://github.com/particle-iot/PowerShield for a good example.
- Upload a private version of your library by running `particle library upload`
- Try adding the library to a project using the Web IDE
- Publish the new public version of the library by running `particle library publish mylibrary` in the CLI
- Push to GitHub, and go celebrate!

Using the Desktop IDE

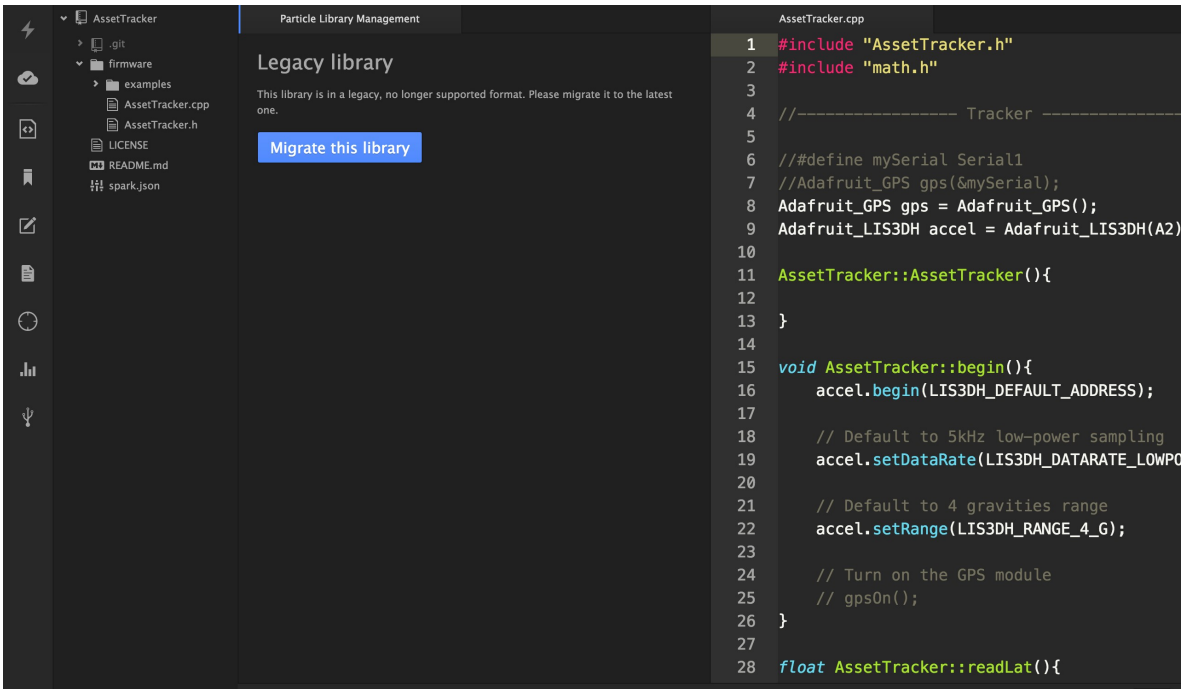
Follow these steps to migrate a v1 Particle library to the new v2 structure using Particle's Desktop IDE:

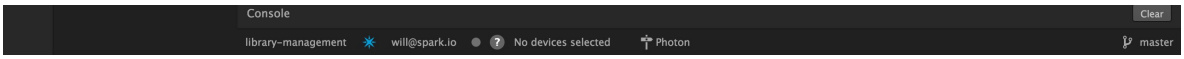
- Install the Particle Desktop IDE or update it to the latest version
    - You can install the Desktop IDE by visiting our download page and selecting the appropriate installer for your computer's operating system
    - If you already have Particle's Desktop IDE installed, you can update it using the following instructions
        - Select `Atom > Preferences` from the top menu
        - When the preferences pane opens, navigate to to `Updates`
        - Click the `Check for Updates` button. This will update your `particle-dev-profiles` and `particle-dev-libraries` packages to the most recent version.

- Open your library directory in a new window of the Desktop IDE
- Open the **`Library Manager`** tab on the left hand navigation bar

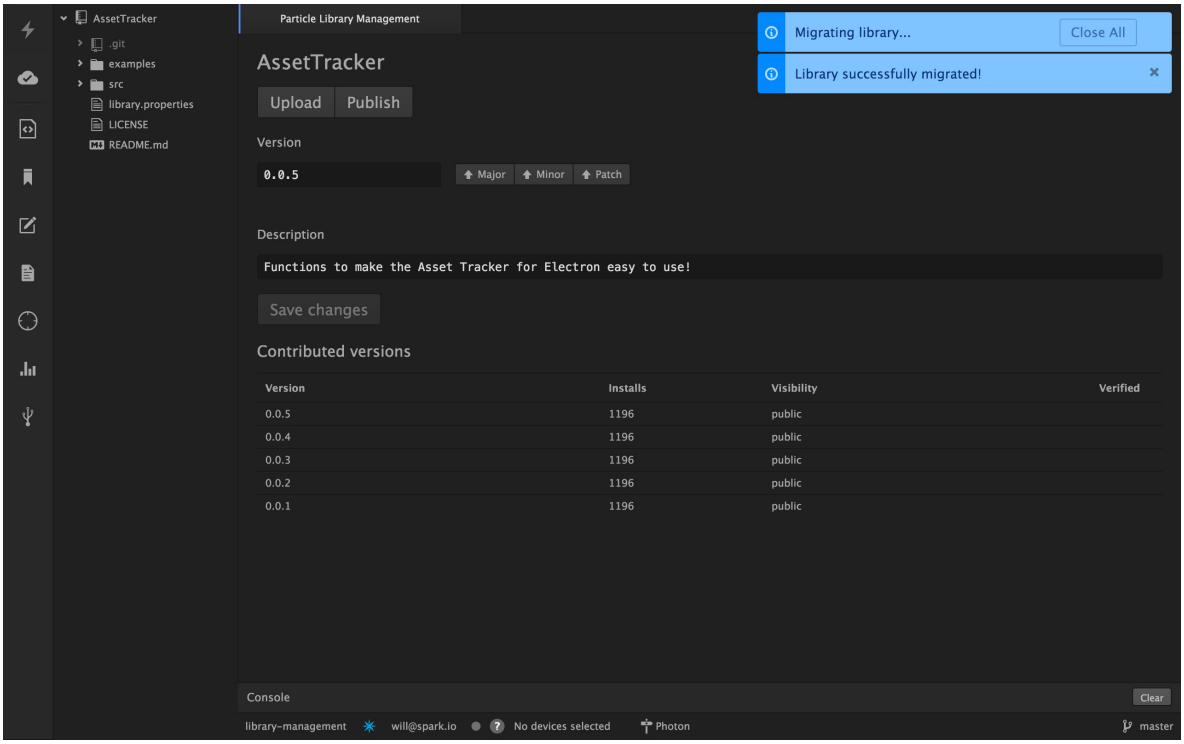


- You will be presented with a notification telling you that you need to migrate your library. Click the **`Migrate`** button





- When migration is complete, you will be notified with a banner alert and presented with new options in the `Library Manager` view



- Edit the newly created `library.properties` file to add a GitHub URL to the `url` field (like https://github.com/particle-iot/internetbutton) and the git remote to the `repository` field (like https://github.com/particle-iot/internetbutton.git)
- Add any necessary external library dependencies in the `library.properties` file as new lines using the following format:

```
dependencies.library_name=0.0.X
```

- Ensure that the example applications for your library compile by opening up an example in the `examples` directory and clicking the compile button
- Refresh the `README.md` file for your library with detailed information and instructions for using and interacting with the library. The `README.md` file will be used as the "home page" for your library.

    - See https://github.com/particle-iot/PowerShield for a good example.
- Upload a private version of your library by clicking the `Upload` button at the top of the Library Manager tab
- Try adding the library to a project using the Web IDE
- Publish the new public version of the library clicking the `Publish` button at the top of the Library manager tab
- Push to GitHub, and go celebrate!

Common issues with migration

- **Include statements**: After you have migrated a library, the process will automatically create a file, `mylibrary/mylibrary.h` that is included for compatibility with old projects. New projects and examples should use `#include "mylibrary.h"`
- **`library upload` scope**: When uploading a new version of a library, *all files in the library directory are uploaded.* Be careful in case you have files in there you don't want to upload like test binaries and large PDFs.
- **Versioning**: You can upload a private version multiple times with the same version number, but once you publish a version to the public you won't be able to upload with the same version number. If you make a small mistake just increase the version number and upload again. Version numbers are free!

If you're having additional issues with library migration or contribution, please feel free to post a message in the `libraries` category of our community forums, or send us a message via our support portal.

SYSTEM FIRMWARE

System firmware is low-level firmware code that supports a Particle device's basic functions. You can think of system firmware as the *operating system* (OS) for Particle's embedded hardware.

Like an operating system for a computer, Particle system firmware provides a foundation for other applications to run on. More specifically, it enables *application firmware* (the firmware you write) to run successfully by exposing the underlying behaviors of the device.

Particle system firmware abstracts much of the complexity away from the traditional firmware development experience. Some specific responsibilities of system firmware include:

- **Secure communication**: Ensuring that all communications between the device and the Particle cloud are authorized and encrypted
- **Hardware abstraction**: Providing a single, unified interface to the device, regardless of the underlying hardware architecture
- **Application enablement**: Exposing a feature-rich API that is used by developers to write applications for the device
- **Over-the-air updates**: Allowing rapid remote changes to code running on the device while providing resilience in poor connectivity environments

Unlike application firmware, system firmware is written and maintained primarily by the Particle team. This is a conscious decision meant to keep you focused on your particular use case without needing to understand the nuances of low-level device behaviors.

That being said, Particle's firmware repository is available as an open source project for those that want deep visibility into system firmware. To view the code and/or contribute, check out the repo on GitHub.

Versioning

New features, security patches, and bugfixes are introduced to new versions of system firmware regularly by the Particle team. These changes to system firmware are bundled into *releases* that are versioned using semantic versioning best practices.

System firmware versions that are suffixed with `-rc.x` are called *prereleases* ("rc" stands for release candidate). These prereleases contain the changes that will eventually become a default release, but still need more thorough usage and testing. We recommend that you **do not** flash prereleased firmware to your production units deployed in the field.

Each release is documented thoroughly to give you a comprehensive picture of what has changed in the new version. For a full list of system firmware releases and their descriptions, please check out the release page on GitHub for the firmware repository.

Firmware Modules

Particle firmware is split into modules: two or more modules for the system firmware and one module for the application firmware.

Each module can be updated independently. This is why over-the-air updates to the user application are so fast: you can update only the user application module without having to update the system firmware modules.

Firmware Dependencies

Application firmware that is written in the Web or Desktop IDEs are *compiled against* a specific version of system firmware before being sent to a device to run. That is, the system firmware acts as a translator - taking the human-readable code you write and translating into a binary that the device is able to run.

This creates a dependency that must be carefully managed. Application firmware can only run on a device with the same or newer system firmware

version than the system firmware used to compile it. This is because the application firmware may be using new functionality that was not available in an older version of system firmware. Allowing a new application to run with older system firmware might lead to a crash.

For example, imagine a new firmware primitive was introduced in system firmware version `1.0.0`, `Particle.travelInTime()`. As an aspiring time traveler, you quickly add the new feature to your firmware logic and send the code off to be compiled and flashed to your Electron.

However, the Electron on your desk is running system firmware `0.9.0`, a version that predates the time travel functionality. Instructing the device to use the new firmware method in application firmware before it understands how to do so will of course not work. You can see how application firmware *depends on* a compatible version of system firmware.

So what happens in these cases?

Safe Mode

When booting up, the Particle device will check dependencies between the application and the system firmware. In the case of an incompatibility between the two, the device will automatically enter into *safe mode* (breathing magenta).

Safe mode allows the device to connect to the Particle cloud, but does not run application firmware. There are many uses for safe mode, but is particularly relevant when a device receives application firmware compiled against a newer version of system firmware than it currently is running. In this case, safe mode prevents the device from running the incompatible application firmware that will cause it to hard fault.

Safe Mode Healer

Safe mode healer takes things one step further, automatically resolving system firmware incompatibilities when a device enters safe mode. Because a device in safe mode still has network connectivity, it is able to send an event to the Particle Cloud notifying it of the firmware mismatch.

From this event, the Particle Cloud determines the cause of the incompatibility and delivers to the device new system firmware over-the-air. When this happens,

you'll see the device rapidly blinking magenta as it receives packets containing the new system firmware. When complete, the device will have what it needs to successfully run the application that was previously flashed.

The combination of safe mode and safe mode healer provides you confidence when flashing new application firmwares to devices. Regardless of the version of system firmware the app was compiled against, the Particle Cloud will ensure the device has what it needs to run it without issue.

A couple of important notes:

- System firmware versions are *backwards compatible*. That is, you can flash an app compiled against an older version of system firmware to the device without the device entering into safe mode. The device will not be automatically downgraded to the older version of system firmware
- System firmware is *modular* and contains multiple parts. That is, your device will receive 2 or more binary files when getting a system firmware update. When receiving system modules via safe mode healer, the device will reset between each binary flashed to it

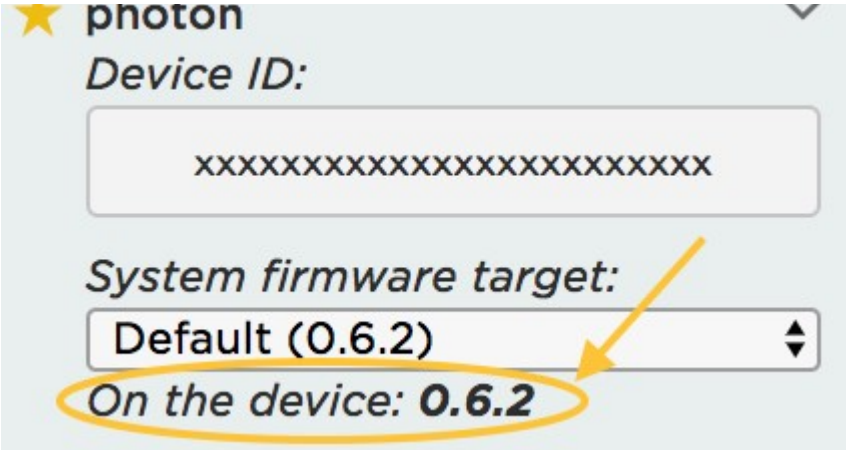
Managing System Firmware

Advanced users may need the ability to actively manage system firmware on a fleet of devices. The management tools needed include:

- Visibility into the version of system firmware running on a device
- Ability to easily update the version of system firmware running on a device

Which version is running on my device?

The easiest place to find the version of system firmware running on your device is in the Web IDE. From the main view, click on the devices icon (◌) from the sidebar. This opens up the device drawer. From here, find the desired device and click on the arrow (>) to expand device details. You should now see the system firmware version running on the device:



*This device is running system firmware version **0.6.2***

You can also find this information in the Desktop IDE, in the bottom rail:

Note that you will need to be the owner of the device to have visibility into system firmware in the IDE. Now that we know the version of system firmware on the device, how can we update it to a different version?

Updating remotely

For devices in which you do not have physical access, you have the ability to update system firmware over-the-air.

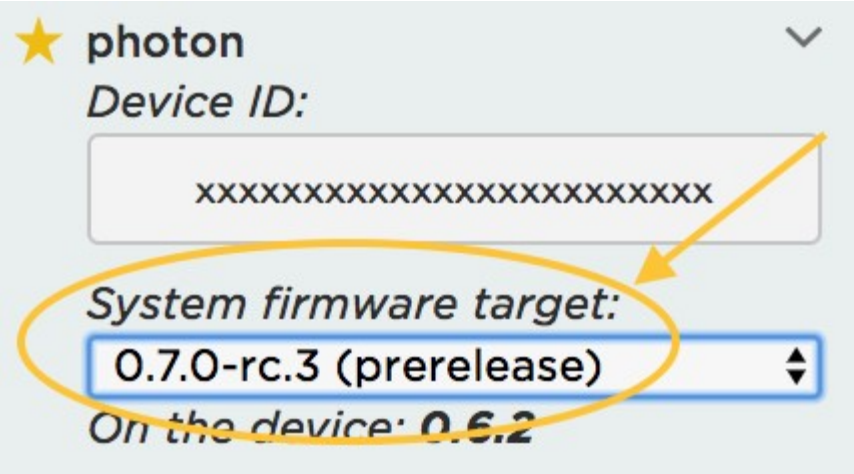
Flashing application firmware

One of the easiest and safest ways to update system firmware is to simply flash an application firmware that was compiled against a newer version of system firmware. The device will receive the incompatible firmware app, and use safe mode healer to automatically download the newer system modules.

Note: This method is not currently available for Electrons, unless the device has been added to a product. This is meant to help you manage cellular data usage.

For updating system firmware on Electrons, check out the methods for updating locally.

In the Web IDE, this can be done by using the *system firmware target* dropdown, and choosing a version that is newer than what is currently on the device.



*In this case, the app will be compiled against **0.7.0-rc.3**, a prereleased system firmware version*

This can be just as easily accomplished using the Desktop IDE:

Now, compile and flash the firmware by clicking on the flash (⚡) icon. Your device will receive the new application firmware and reboot. Then, it will automatically enter safe mode and trigger the cloud to resolve the incompatibility by sending it system firmware version 0.7.0-rc.3.

Sweet! You just updated the system firmware on your device.

There's a couple of things to note:

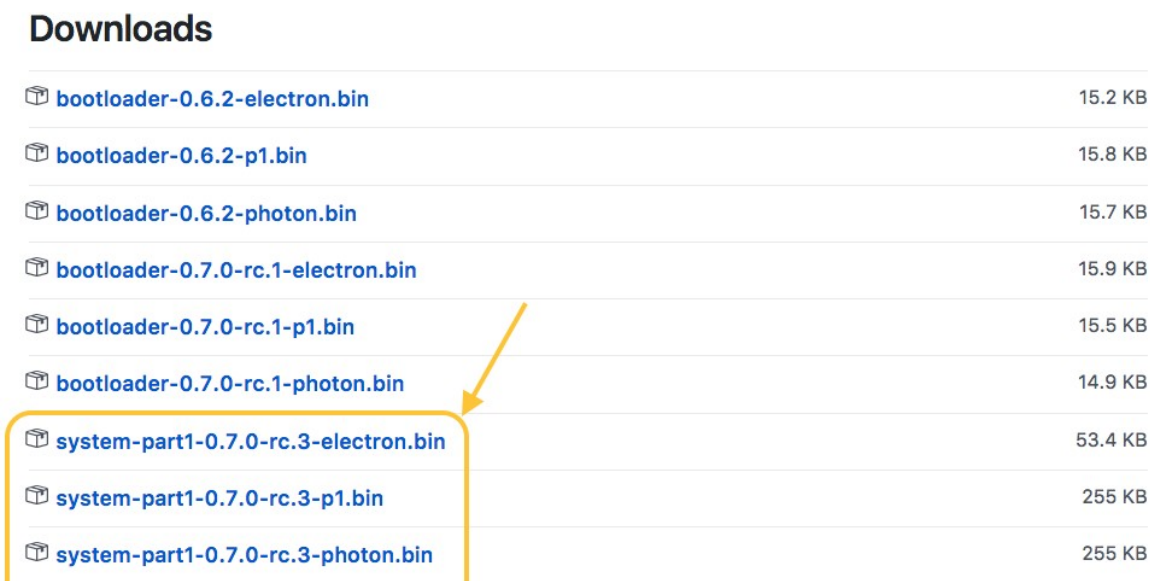
- This approach will also work for *product firmware*. When a product firmware binary is released to a fleet, any device that receives it will enter into safe mode and heal itself by downloading the required system firmware
- This approach will trigger system firmware *upgrades*, but not *downgrades*. As mentioned earlier, system firmware is backwards compatible meaning that devices can successfully run application firmware compiled against an older version of system firmware than it currently is running

CLI (Remote)

You can also use the Particle CLI to remotely update a device's system firmware without changing the application firmware. This is a more advanced approach and requires some technical chops.

To do this, first visit the system firmware releases page on GitHub and locate the version you'd like to send to a device.

When you find the desired release, scroll down to the **Downloads** section. Here you will find the system firmware binary files. Remember that these binaries are specific to a device type, and a complete system firmware is comprised of multiple parts. Hone in on the files that begin with `system-part` :



Available downloads for the 0.7.0-rc.3 release

Find the files relevant to your device (each binary is suffixed with the device type) and click to download them to your machine. Note that you'll only need to do this step once to store a copy of the binaries on your computer.

Next, you'll flash these files to a device using the `particle flash` command in the CLI. If you haven't already, you must download the Particle CLI. Open up your Terminal and run the following commands to flash the system modules to a device:

```
particle flash YOUR_DEVICE_NAME_OR_ID path/to/system-part1.bin
particle flash YOUR_DEVICE_NAME_ID path/to/system-part2.bin
# Sometimes required
particle flash YOUR_DEVICE_NAME_ID path/to/system-part3.bin
```

Use caution when using this method. Double check that you are flashing the correct binaries for the given device type, and that you flash all required system parts.

Updating locally

For devices in which you have physical access, there are also methods to update system firmware over-the-wire.

CLI (Local)

The Particle CLI offers two different methods of updating system firmware locally. Both require that the device is connected to your computer over USB. If you haven't already, you must download the Particle CLI and ensure you are running version **1.24.1** or later. You can check with `particle --version`.

The first approach is to run `particle update`. Open up your Terminal and run the following command to flash the latest system firmware to a device:

```
# put the device in DFU mode first, then update the system firmware
$ particle update
> Your device is ready for a system update.
> This process should take about 30 seconds. Here goes!

! System firmware update successfully completed!

> Your device should now restart automatically.
```

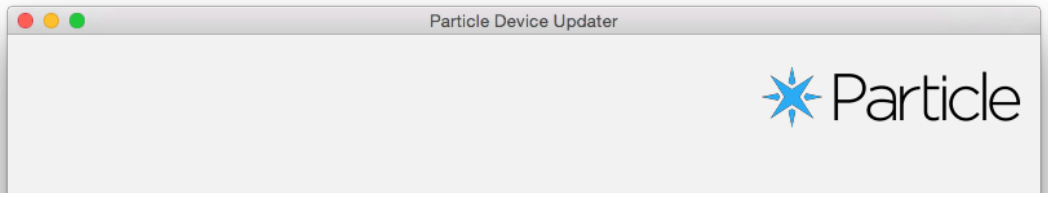
Be sure to put the device in DFU mode before running the command. Note that this will update your device to the *newest* system firmware - it does not currently allow you to flash a different verrsion of firmware other than the latest.

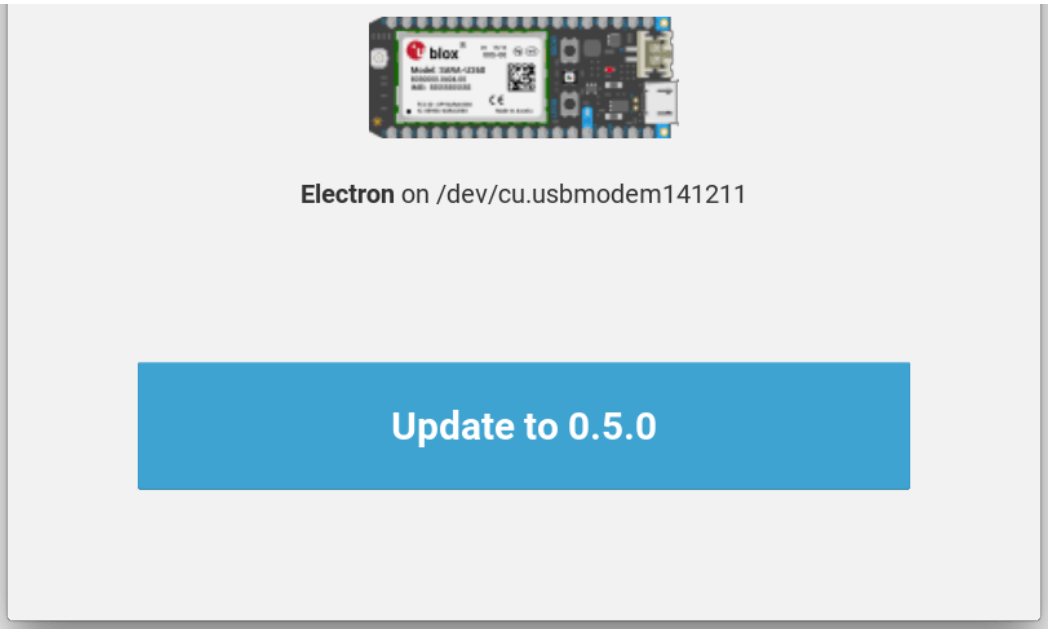
If you'd like to use the CLI to flash a system firmware version *other than the latest*, you can use the `particle flash` command in a similar way as outlined above. The only difference will be that you'll pass an argument to tell the CLI to flash the files over USB, and you won't have to include the device name or ID in the command:

```
particle flash --usb path/to/system-part1.bin
particle flash --usb path/to/system-part2.bin
# Sometimes required
particle flash --usb path/to/system-part3.bin
```

Firmware Manger

The Firmware Manager is a desktop application that upgrades your device to the latest system firmware. For Electrons, it provides an easy way to update system firmware while avoiding cellular data charges.





The Firmware Manager is available for Windows and Mac

Like the `particle update` command, the Firmware Manager updates your device to the *newest* system firmware. For more information on this method, please check out the firmware manager guide.

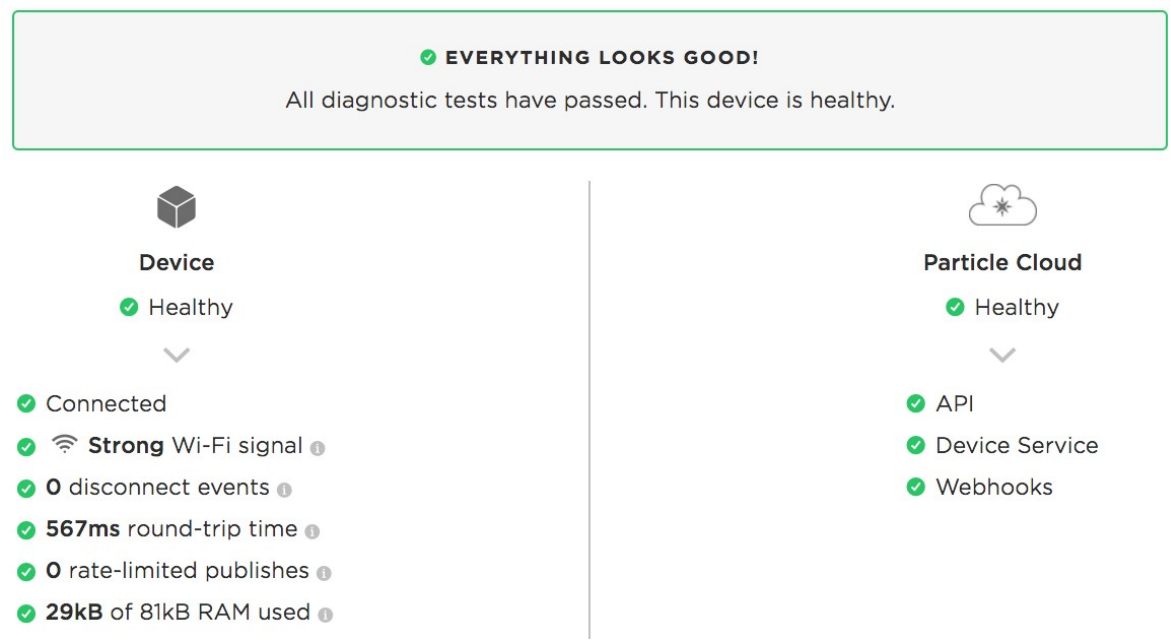
REMOTE DIAGNOSTICS

Beta feature

*This feature is **currently in private beta**. If you would like access to this feature, please contact us.*

As you deploy your IoT fleet into the field, it becomes increasingly important to ensure that devices stay in a healthy state. In addition, when problems do arise, the ability to quickly identify and implement a solution are paramount to the reliability of your production deployment. This level of visibility can often be difficult to achieve without physical access to these devices in the field.

Remote Diagnostics gives your team the power to actively monitor the health of deployed units from the Console, without the need for custom development or costly dispatching of technicians. Furthermore, in the event of a device falling into an unhealthy state, your team will be empowered with rich context and suggested courses of action to quickly diagnose and rapidly resolve the issue.



Remote Diagnostics allow you to test the connectivity health of your devices, and quickly resolve problems when they arise.

Connectivity Layers

Multiple connectivity layers must be operating successfully for a given device to be able to successfully communicate with the Particle Cloud. Note that the relevant connectivity layers vary based on the type of device (i.e. Wi-Fi vs. Cellular).

These connectivity layers are:

Device

The device itself must be in a healthy state in order to successfully communicate with the cloud. A variety of factors influence its state, such as battery state of charge, signal strength, available memory, and application firmware that does not exceed enforced rate limits.

Particle Cloud

The health of the Particle Cloud is critical to devices having the ability to successfully connect and communicate. There are a few Particle Cloud services in particular that directly impact device health and communications:

Device Service

The Device Service brokers the connection between an IoT device and the Particle Cloud. In addition, the Device Service is responsible for shuttling of messages to and from a Particle device.

API

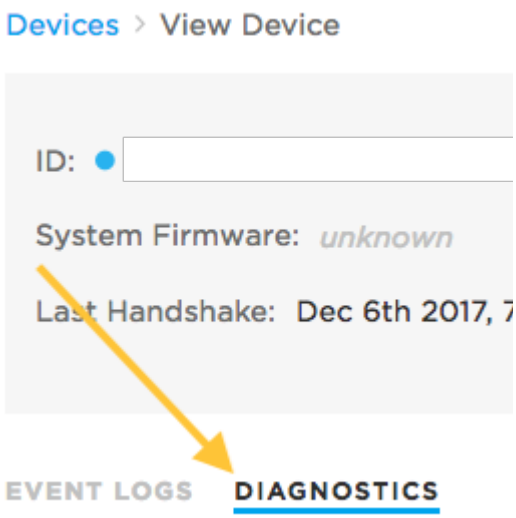
The API provides a REST interface to allow remote interactions with Particle devices and the cloud. This includes calling a function, checking a variable, or publishing an event that devices subscribe to.

Webhooks

The Webhooks service allows for device data to be sent to other apps and services. Webhooks also allows devices to ingest information *from* these Internet services.

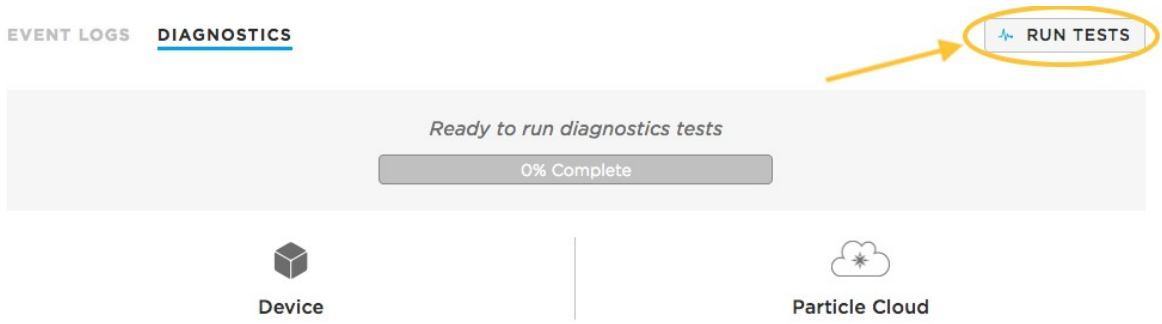
Running Diagnostic Tests

Diagnostics tests can be run for a device using the Particle Console. To access Remote Diagnostics, click on a device from your device list (on the devices view) to visit the device details page. From here, click on the Diagnostics tab. This toggles between the Event Logs and Remote Diagnostic tests for the device.



Remote Diagnostics are available on the Console's device details page. Click on the Diagnostics tab to get started.

You are now presented with the relevant connectivity layers as described above. Click the **Run Tests** button to trigger the execution of a variety of health checks:



Running the tests will kick off diagnostics for each layer of the connectivity stack. Tests will be run in parallel, and the test results will be shown once all tests are completed. Let's dive into what each test actually does:

Device

Starting with system firmware version `0.8.x`, Particle devices have the ability to collect a rich amount of diagnostic data and send this information to the Particle Cloud.

Device diagnostics are sent to the cloud at two different times:

- Automatically, when the device *handshakes* (starts a new secure session with the Particle cloud)
- On-demand, when the diagnostic tests are run in the Console or via the API

The device collects the following diagnostic vitals:

- *Signal strength*: The strength of the device's connection to the Wi-Fi network, measured in decibels of received signal power.
- *Disconnect events*: The number of times the device disconnected unexpectedly from the Particle Cloud since its last reset.
- *Round-trip time*: The amount of time it takes for the device to successfully respond to a CoAP message sent by the Particle Cloud in milliseconds.

- *Rate-limited publishes*: Particle devices are allowed to publish an average of 1 event per second in application firmware. Publishing at a rate higher than this will result in rate limiting of events.
- *Used Memory*: The amount of memory used by the device, combining the heap and the user application's static RAM in bytes.

The device delivers the diagnostics data to the Particle Cloud via a system event that is published to the event stream. The device diagnostic event will have the name `spark/device/diagnostics/update`, and include a data payload of the most recent diagnostic vitals the device collected.

To ensure that your device is able to collect and send diagnostic data to the Particle Cloud, you will need to ensure that the device is running a system firmware version equal to or greater than `0.8.0`. For information on managing system firmware, check out the system firmware guide.

Particle Cloud

When running the test suite, the Particle Cloud services most relevant to device connectivity are automatically checked to ensure they are fully operational. Probed services are:

- Device Service
- API
- Webhooks

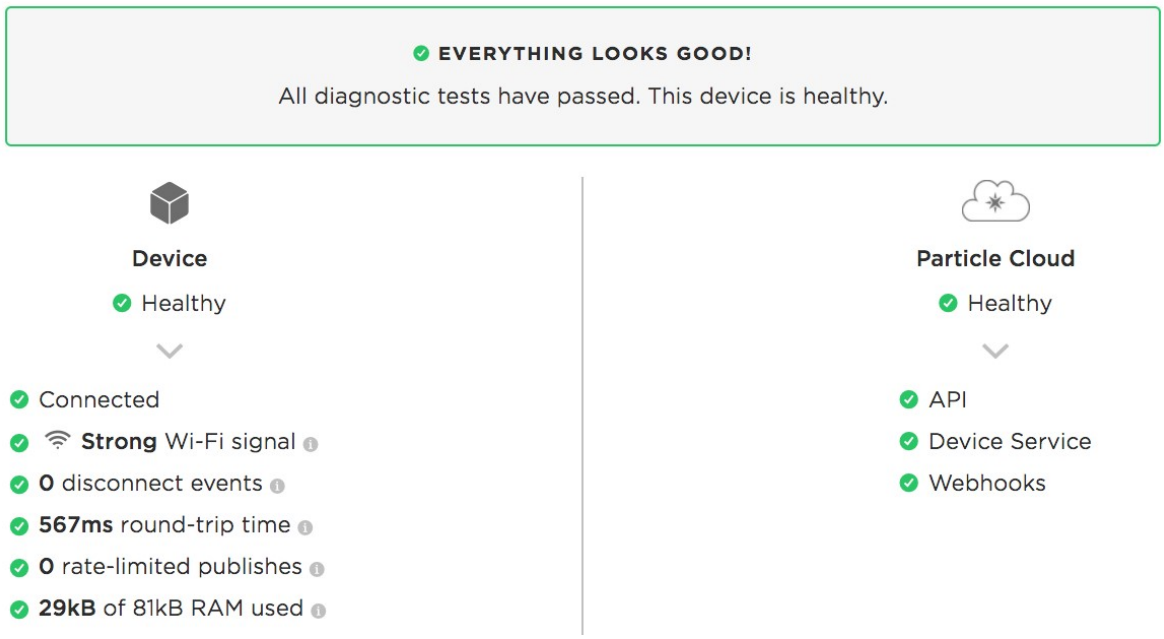
This test is made possible by a tight integration with Particle's status page. Any open incident involving the services above will be reflected in the test results.

Test Results

Once all of the diagnostic tests have completed, the Console will provide test results. Each connectivity layer will marked as *healthy*, *unhealthy*, or *warning* depending on the result of the test.

Healthy

A *healthy* test result means that all tests have passed successfully. The device is operating normally. This state looks like this:

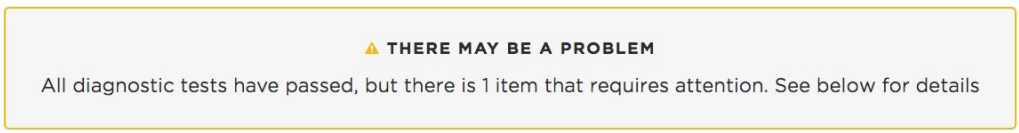


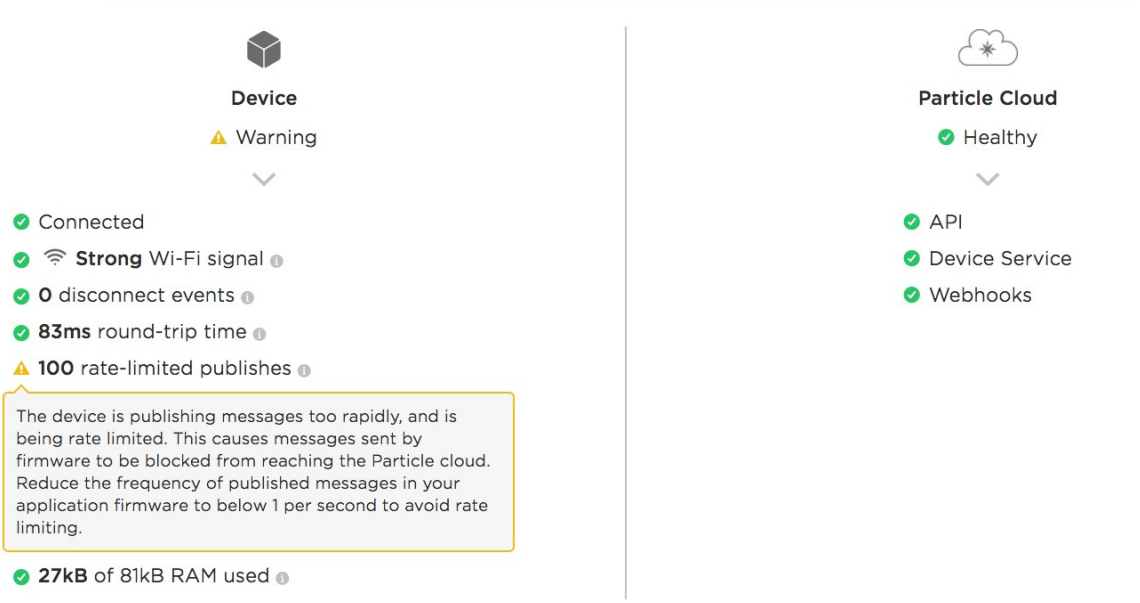
All diagnostic tests have passed and this device is healthy! Woot!

You can see that each connectivity layer has been marked as *healthy*, with a green checkmark. You will also notice a top-level summary that confirms that all tests have passed and diagnostic vitals are in healthy ranges.

Warning

The diagnostic tests also can be marked in the *warning* state. In this case, one or more of the diagnostic vitals has fallen outside of the healthy range. However, all diagnostic tests still passed. This is an indication that there *may be a problem*, and you should investigate it further:



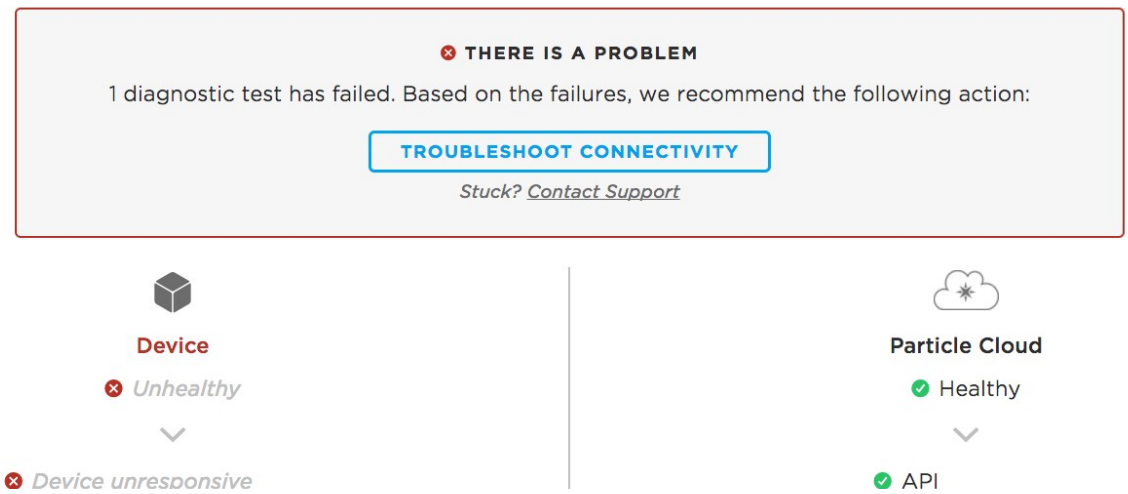


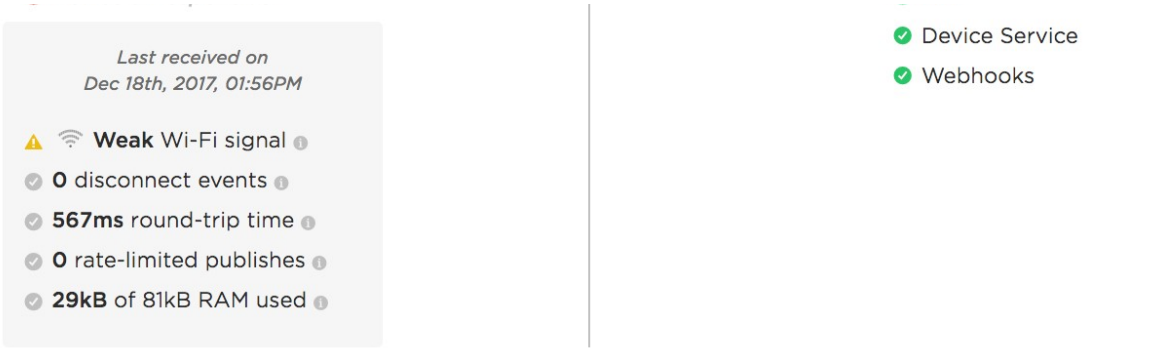
In the warning state, you will receive some helpul text to explain what is happening as well as some recommendations on how to return the device to a fully healthy state.

In this case, the device is getting rate-limited in firmware because it is attempting to publish events too quickly. The recommendation is to rework the application firmware by reducing the frequency of event publishes to 1 per second or less.

Unhealthy

The test run will be marked as *unhealthy* if one or more of the Remote Diagnostic tests fail. Note that failure is defined as a state in which the device will not be able to communicate with the Particle Cloud:





This Remote Diagnostic test reports a problem because the device is unresponsive

In this state, the test will be marked clearly as failing with a red "X" icon. In this case, we are not able to successfully communicate with the Particle device. The device layer is marked as unhealthy, and we see that the device is unresponsive.

Anytime the Remote Diagnostic tests fail, there will be a course of action suggested in the test results summary. These calls-to-action are designed to help your team quickly identify a solution to the connectivity issue that has arisen. In this scenario, the call to action is to visit the docs to troubleshoot device connectivity. Remote Diagnostics provides this call-to-action intelligently based on the test failures.

To help uncover what the cause of the issue might be, the last known device diagnostic reading is displayed. For this device, we can see that the last known diagnostics reading showed a weak Wi-Fi signal. This may be the cause of why the device is now not responsive to requests from the Particle Cloud.