For all of these articles I had a brief video call with an engineer, wherein they told me about their work, I asked questions and took notes, and then wrote the pieces. They were edited in collaboration with the engineer and my manager and published to the site:

Electron, Desktop Development at the Speed of Electricity

By Shane Hathaway ghostwritten by Aisling Fae August 24, 2020

If you have used Electron before, your initial experiences may have been less than ideal. Perhaps the apps you used were slow and clunky. However, did you know that Slack and VS Code are both Electron apps? They run pretty smoothly. Perhaps it is time to give it a second (or first, or third) look!

Please Meet Electron

Powering Stand-Alone Web Apps

Electron is an open source framework for developing desktop applications using web development technologies. It is basically a way for web apps that you would run on a website to run stand-alone in their own window, in their own environment.

An Electron app consists of two main parts: the main process and the various renderer processes. The code for the main process is written in Node.js. The code for each rendering processes is written in HTML, CSS, and JavaScript, just like a web page. You can actually use web development frameworks like Vue.js or React to write your rendering processes.

Super Fast Turnaround

Electron's number one strength is its turnaround speed. No other application development framework can go from 0 to fully functioning app as quickly as Electron can. Recently we were able to turnaround an app for a client in 2 weeks, because it was built on top of an existing React library. React, being a web development framework, integrates with Electron seamlessly.

Beginner Friendly

Another advantage of Electron is that it's super beginner friendly. The amount of knowledge of low-level libraries one would need to acquire to program a similar application in, say, C++ or Python, and have a nice looking user interface is daunting to say the least. With Electron, if you've ever written a web page, you can write an Electron app.

Cross-Platform

And why would you want an Electron app instead of just a webpage? Electron makes it so that the app environment is consistent. It handles the communication to the OS for you, so it is super simple to export installable packages for Windows, Mac, and Linux. It's as simple as running `yarn make`. It also cuts down on your support costs because you won't have to answer a thousand and one emails with "please use the latest version of Chrome".

Quick Start

Electron Forge is an open source tool for getting started with Electron quickly. If you already have the *yarn* development tool installed, all you need to type to start creating your app is:

```
yarn create electron-app my-new-app --template=webpack
```

If you're a fan of TypeScript (an enhancement of JavaScript that adds static type checking), use this alternative:

```
yarn create electron-app my-new-app --template=typescript-webpack
```

Once the app is created, *yarn start* it. You'll see a "Hello World" app and the familiar Chrome development tools. Edit the code with your favorite editor.

You might wonder why Electron needs Webpack. Webpack is the de facto way to write modern JavaScript and compile your code to a representation that works in many environments. If you want to use frameworks like React in Electron, Webpack is the quickest way forward.

The Not-So-Good About Electron

The ease of development and streamlined cross-platforming do come at a cost.

Resource Hog

Electron apps used to be so slow they would turn people off the technology. They've gotten much faster, but they're still slower and more resource

intensive than other native apps. On most rigs, that won't matter, but on small laptops or netbooks, they can cause a measurable drain on memory, processor, and battery usage.

Too Many Batteries Included

The installable package that Electron generates tends to run on the large size for simple applications. You're looking at about a 60 MB minimum with default settings. Not a lot for a big software suite, but maybe too much for a simple, single use applet. This is because, with Electron, you're kind of throwing everything plus the kitchen sink along with your app. Any Electron app comes with its own implementation of the Chromium engine, a sort of proto-web-browser. So there will be a lot of things included with your app that your app itself might not use, but trade-offs is the name of the game with modern software development.

Go Forth and Electron

One of the most alluring features of Electron is its beginner friendliness. If you've never developed a GUI desktop app before, but you know HTML, CSS, and JavaScript, you can go to electronforge.io right now and quickly find yourself `yarn make`-ing your own app. If you don't have any web development experience, start by picking up the basics of HTML, CSS, JavaScript, and Node.js.

If you have written a lot of programs that run on the command line and always wanted to have a program with a window that users could click on, Electron gives you that chance. It delivers on the same promise that Borland Delphi or Microsoft Visual Basic delivered on back in the 90s: the promise of the window, of the graphical user interface. Only this time, it runs on any OS on any machine. Go forth and get your window; Electron makes it easy.

4 Reasons to go with Python over ${language}

By Calvin Hendryx-Parker ghostwritten by Aisling Fae October 28, 2020

“Use the right tool for the job” is a golden rule of engineering. Working on software, however, there are often dozens of different languages that can get the job done. How can you know which one constitutes the right tool for the job? Some programming languages are designed with one use in mind, but most try to be general purpose. One language that stands out for being general purpose and yet often out-performing languages custom built for the task is Python

In this post we’ll look at some of the reasons to choose Python whenever it's in competition with any other language for the "right tool for the job” status.

1. "Python is the second-best language for everything"

This was actually a quote from Dan Callahan, speaking at PyCon 2018. If it's the second-best language, why not use the first-best language? Simple — because then you’d have to learn a whole new language. Unless the gulf from second to first best is significant enough – if the custom “made for the task” language provides indispensable features and performance improvements – it simply does not pay to learn a new language when Python more than suffices.

When you work on a truly diverse set of problems, it helps to have a complete mastery of a language versatile enough to tackle any and all of them. For me, that language is Python.

2. Batteries Included

You can go a long way writing Python before you have to download some external dependencies to make your code do something useful. Node.js and JavaScript are an example of the opposite — if you need to do anything there, you are npm install 'ing something and now you are down a road of tracking those dependencies and dealing with when they break. Not so with Python. The Python Standard Library is expansive and robust enough to tackle most problems.

Here is an amazing example of "batteries included" at work in the wild. One of my favorite conference speakers, David Beazley, got a gig doing eDiscovery for a legal case. He had to sit in a secure room and process 1.5 TB worth of source code looking for certain patterns. All he had to do it with was a

computer with a fresh OS install, and no Internet connection. He had no way of bringing in additional programs or downloading any libraries, but the computer happened to have Python installed. So, he wrote his own set of tools on the spot to aid in sealing the deal on that case.

You're not likely to find yourself in quite the same extreme circumstance, but you will still benefit from minimizing dependency management and building more robust applications.

3. The Python Community

It's hard to overstate what a blessing the Python community is. Writing Python puts you in a community with a vibrant, diverse, and enthusiastic cadre of developers, designers, scientists, and more. You can attend multiple conferences all over the world and meet all the exciting people working to make Python as great as it is. I should know, as I have attended 16 PyCons (in a row!) and 5 EuroPythons, plus many regional events such as PyOhio and PyTennessee.

If you write Python, you'll have by default a huge pool of information and resources to educate yourself and write better software. Additionally, if you decide you do want to avail yourself of external libraries, Python has over 100,000 of them. If there's something you can't do with the Python Standard Library, chances are there's a library out there that adds that capability (or one is being written). You can contribute to these libraries yourself, and if you want to write your own you can get a lot of support from the community.

4. Beginner Friendly but Ready for the Show

There is a good reason that Python is the number one teaching language on this planet: it is easy to get a new person started. There isn't a lot of boilerplate or ceremony to build a program that can actually do useful work.

If you work at a company that's growing – therefore, constantly hiring new talent – running a Python codebase means virtually every employee you hire will already have some experience with the language. This cuts down on training cost and time significantly. Your junior engineers can go from hired to writing code in record pace.

Then, on the other end of the equation, you can look to applications like Instagram, which uses the Django Python web framework to serve a tremendous amount of requests each day. For tasks like this you can rely on features such as static type hints to ensure your code is bulletproof and more easily testable. Here is where your senior engineers get to flex and build Object Oriented models and optimize performance.

In summary

These are the 4 biggest reasons why I keep coming back to Python after all of this years. Ultimately, every project's requirements will dictate which tool is the right tool. However, if Python is an option, and statistically it almost always is, it's an option you should consider.

Introduction to Infrastructure as Code with Terraform

By Caleb Gosnell ghostwritten by Aisling Fae October 13, 2020

Terraform, by HashiCorp, is a very useful tool in the arsenal of any seasoned DevOps or cloud infrastructure engineer. It is an Infrastructure as Code (IaC) tool, which can be used to define and manage resources from a variety of local and cloud service providers. It is useful for maintaining repeatable, understandable, and consistent infrastructure.

Terraform accomplishes this by converting its declarative configuration files, which describe the desired shape and state of the infrastructure and application, into sets of API calls which it can execute to make that state a reality. It saves the results of those calls as "state," which it uses on subsequent runs to determine if changes are needed. Put simply, you tell it what sort of infrastructure and resources you want, and Terraform goes and sets them up for you.

Let's take a closer look at how to use Terraform, and how it converts code into infrastructure:

Command and Configure

To start using Terraform (after installing it and configuring it for your providers) you will write a configuration file. Terraform Configuration files have extension `.tf` and are written using the HashiCorp Configuration Language syntax, which is “designed to be easy for humans to read and write” (it's also possible to use straight JSON).

```
provider "aws" {
  profile = "default"
  region  = "us-west-2"
}
resource "aws_instance" "example" {
  ami           = "ami-830c94e3"
  instance_type = "t2.micro"
}
```

This Terraform configuration specifies that there should be an AWS Instance in the region `us-west-2` with an instance type `t2.micro` built using the AMI `ami-830c94e3`. The credentials Terraform should use in issuing the API calls to verify and create this instance can be found using the `[default]` section of the current user's `~/.aws/config`. Alternatively, this could be configured to use environment variables, or assumed roles.

Before Terraform can spin this up for you, you must run `terraform init` which caches provider applications and module configs into `.terraform`. Terraform will complain if you don't run this command first. All Terraform commands should be executed in the directory which has the Terraform Configuration Files you wish to use.

Once that's taken care of, you can run `terraform plan` and Terraform will reach out to AWS (or your chosen remote provider), figure out what you already have running there and compare it with what you want. It will then inform you of what changes it needs to make. If you detect any errors at this point, simply change your config files and run `terraform plan` again. If you're ready to go run `terraform apply`.

`terraform apply` will first give you the same preview as plan did and then asks if you want to go ahead and apply these changes. Respond `yes` and Terraform will execute the plan.

> A note on best practices:
>
> While it might be tempting to skip the step of running `plan`, since you get the same information from `apply` with an option to cancel the changes, it is strongly recommended to use `plan` consistently while making changes to the configuration files, and only run `apply` when you're relatively certain you're ready to pull the trigger. There is nothing quite like the headache of a mistakenly applied set of Terraform changes.

Once your resources are live, you can change your configuration file at any time, and run `terraform plan` to see changes and `terraform apply` to execute the desired changes. Terraform does not run on the background — like Salt, Puppet, or Chef — it runs only when invoked. Changes to your configuration file, and therefore your infrastructure, can and should be tracked using version control.

If you want to spin down all of your resources, you can run `terraform destroy`. This works like `terraform apply`, as it will tell you all the resources it plans to destroy and prompts you with `yes` or `no`.

The Terraform State File and Other Useful Commands

By default, Terraform keeps a state file locally called `terraform.tfstate`. Its primary function is to store bindings between the resources declared in your configuration and the resources configured in your providers (e.g. AWS, GCP, Datadog). It stores references to the remote objects, addresses, and other metadata. Basically, `terraform.tfstate` contains everything it will need to update that resource on a later date, as well as information about it that might be useful to the engineer. When teams are using Terraform, it is typical to configure the state data to be written to a cloud storage service so that the latest information is always readily available.

You can probe the state file using `terraform state`:

- `terraform state list`: list items stored in Terraform's state
- `terraform state show <resource_address>`: print details of item from Terraform's state
- `terraform import <resource_address> <identifier>`: bring an existing resource under Terraform's control. You can do this with a minimal resource config, then use `terraform plan` or `terraform state show` to see about fleshing it out.
- `terraform state rm <resource_address>`: remove a resource from Terraform's control without changing it. You are then free to remove the Terraform config for the resource, and Terraform will not try to destroy it.

Similarly, if you keep the config for the resource, Terraform will want to make a new one.

- `terraform console`: interact with the interpolation parser without running plan or apply. This is quite useful for working out tricky syntax issues.

Terraform refreshes the state information from resource Providers when you run Terraform operations, so it's always up to date before applying any Terraform config file.

```
$ terraform state show 'packet_device.worker'
# packet_device.worker:
resource "packet_device" "worker" {
    billing_cycle =  "hourly"
    created       =  "2015-12-17T00:06:56Z"
    facility      =  "ewr1"
    hostname      =  "prod-xyz01"
    id            =  "6015bg2b-b8c4-4925-aad2-f0671d5d3b13"
    locked        = false
}
```

Output from an example `terraform state show` query.

Main Takeaways

This should give you a pretty good primer on what Terraform is and how to use it. For more information, do peruse their extensive documentation, as well as the getting started guides. Now that the basics are out of the way, we'll be expanding on this series with posts on more obscure features, tips, and tricks for Terraform. Keep an eye out for those.

What's New In Plone 6

By Kim Nguyen ghostwritten by Aisling Fae September 18, 2020

Long time followers of the Six Feet Up Blog know all about the open source content management system Plone. Indeed we have published many Plone articles on this site, such as why you should upgrade to Plone 5 or how Plone handles permission management. Some posts are over a decade old now –for instance this article on Plone 4 from 2010– and you can also find a general introduction to Plone on our website.

This post, however, is about the future, specifically the upcoming major release of Plone 6.0. The main features of this release will be the migration to Python 3, the Plone Rest API, and Volto, the brand new React.js based front-end.

"But wait!" you say. "Aren't all of those features already in Plone 5.2?" That's correct: all of these exist in some capacity in Plone 5.2 that was released in 2019. This is because Plone 5.2 was a special release of Plone developed in parallel with Plone 5.1. This made it possible to get a version of Plone running on Python 3 as quickly as possible, since Python 2.7 officially reached its end of life on December 31, 2019.

The Volto front-end is a free and open source framework that can be used for Plone. The Plone REST API has existed as an add-on for Plone since at least 2015 and was added to the core for Plone 5.2.

However, Plone 6 will be the first release in which all 3 are core features of Plone, making Plone 6 projects much easier to configure, customize, and maintain.

Plone 6 + Python 3

Plone's migration to Python 3 was a long time coming, as the bell had been tolling for Python 2.7 for years. Plone consists of well over a million lines of code, so porting it to Python 3 was a huge effort by many contributors, who have also produced upgrade guides and recipes.

One of the things that had to come together for this migration was the release of Zope 4. Zope is a free and open source web application server that has been at the core of Plone from the beginning. With the release of Zope 4, also written in Python 3, it became possible to complete the process of migrating Plone to Python 3. Although no major security issues are currently known to exist in Python 2.7, it won't receive further official security updates, and so it was necessary to move on.

Plone REST API

In 2014, at the Bristol Plone Conference, a strategic summit concluded that the future of Plone relied in part on the creation of an interface layer separating the back-end and the front-end of Plone. The resulting plone.api package allowed developers to innovate by decoupling these two parts of the Plone application.

The following year, the Plone REST API was created as an add-on for Plone 5. As with all major Plone innovations, including the Dexterity content type framework and the Diazo theming engine, the REST API was used as an add-on in production systems and eventually was added to core, as of Plone 5.2.

The Volto front-end relies on the REST API to communicate with the Plone back-end, as do other Plone front-ends written using Angular and Vue.js.

Here is an example REST API call to retrieve the contents front page of a site:

```
$ curl -i https://plonedemo.kitconcept.com -H "Accept: application/json"
```

which results in the following JSON output:

```
HTTP/2 200
server: nginx
date: Wed, 16 Sep 2020 15:02:30 GMT
content-type: application/json
content-length: 743
vary: Accept-Encoding
x-frame-options: SAMEORIGIN
```


```
{
  "@components": {
    "breadcrumbs": {
      "@id": "https://plonedemo.kitconcept.com/@breadcrumbs"
    },
    "navigation": {
      "@id": "https://plonedemo.kitconcept.com/@navigation"
    }
  },
  "@id": "https://plonedemo.kitconcept.com/",
  "@type": "Plone Site",
  "id": "Plone",
  "is_folderish": true,
  "items": [
    {
      "@id": "https://plonedemo.kitconcept.com/en",
      "@type": "LRF",
      "description": "",
      "review_state": "published",
      "title": "English"
    },
    {
      "@id": "https://plonedemo.kitconcept.com/de",
      "@type": "LRF",
      "description": "",
      "review_state": "published",
      "title": "Deutsch"
    }
  ],
  "items_total": 2,
  "parent": {},
```

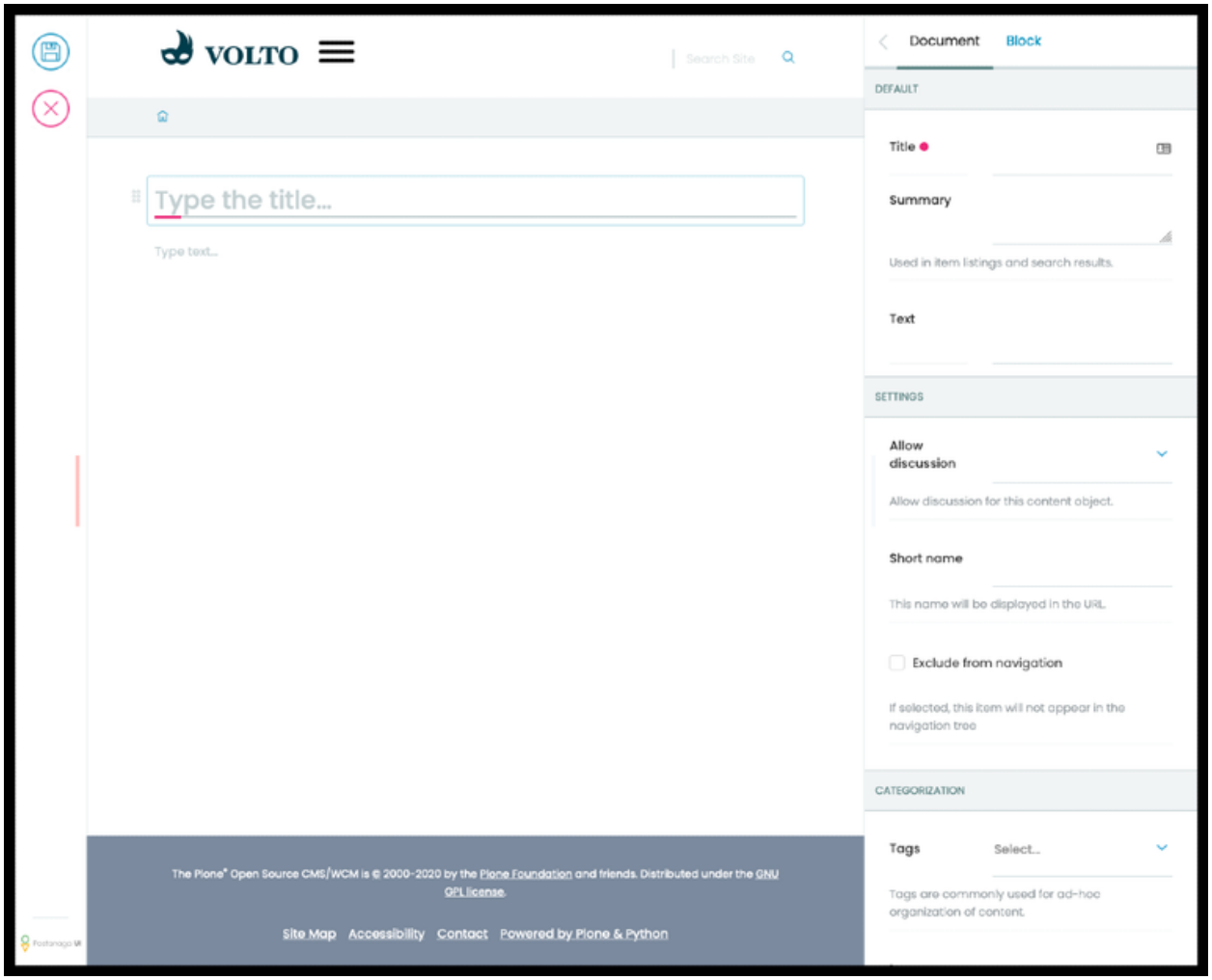
```
  "title": ""
```

This type of communication reduces significantly the network traffic and computation required, compared to the classic back-end rendering of HTML.

React on the Front-End

Plone is a ready-to-use content management system with a complete user interface. Since 2015, with the release of the REST API add-on, Plone has supported other front-ends, built using popular JavaScript tools such as Angular, Vue, Gatsby, and React. The latter, arguably the most popular JavaScript front end framework, is what Volto is built with. As with other core features of Plone, Volto started as a standalone project but, when Plone 6 releases, it will be the default front-end for Plone, coexisting with what we are now referring to as the “Plone classic” UI.

With Volto, content contributors and managers will experience incredibly fast response times on the admin interface, as will viewers on the user side. This will be especially true on mobile devices where the benefits of the REST API’s reduced network traffic will result in near-instantaneous responses. The Volto front-end is much less resource intensive and just plain faster, even on powerful desktop computers. And – don't worry – the Plone classic front-end will still be available, whether you choose to expose it to end users or reserve it for content editors or site administrators.



Adding a New Page with the Volto front-end

Simplified Development

For Plone developers and power users, there's another tremendous advantage to switching to Volto. In the past, altering the core functionality of Plone—in order to build custom templates or add-ons— was quite complex, requiring knowledge of ZCML (the Zope Configuration Markup Language), XML, CSS, and the functioning of over a dozen moving parts.

With Volto, you only need 2 things: knowledge of React.js, and knowledge of the Semantic UI, a highly customizable theming framework. Anyone who knows React will be able to jump into customizing Volto, opening up Plone to a much bigger community of JavaScript developers. With a much lower barrier to entry for Volto developers, there will be a lot more custom features for Plone—from themes, to templates, to add-ons— adding to Plone's vibrant community of users, developers, hobbyists, and professional service providers.

As an example, customizing the basic header styling of a Volto site is as simple as overriding the Semantic UI theme's variables, of which there are over 3000:

```
.ui.basic.segment.header-wrapper {
  background-color: #191919;
  border-bottom: 1px solid #939393;
  margin-bottom: 20px;
}

.ui.basic.segment .header .logo-nav-wrapper {
  justify-content: space-between;
}

.logo .ui.image {
  height: 50px;
}
```

Overriding the header requires adding a new Header.jsx customization file:

```
import { Logo, Navigation } from '@plone/volto/components';

...

render() {
  return (
    <Segment basic className="header-wrapper" role="banner">
      <Container>
        <div className="header">
          <div className="logo-nav-wrapper">
            <div className="logo">
              <Logo />
```

```
            </div>
            <Navigation pathname={this.props.pathname} />
          </div>
        </div>
      </Container>
    </Segment>
  );
}
```

This is so much easier and welcoming than the many technologies required to make the same change in Plone classic UI.

Going Further…

There are dozens of smaller features and tweaks that have been in the works since the last Plone 6 Sprint in April 2020. The 2020 Plone Conference (a virtual event) is scheduled for November 16-22, 2020 and there's likely to be at least an Alpha release of Plone 6 by then.

As with any big technology changes, there is a bit of a learning curve getting developers from Plone classic to Volto. Fortunately, anyone can try Volto and enterprising developers can get a head start today by following along the Volto training materials (the main Volto class, the shorter hands-on class, and Plone's intro to React, Redux, and React-Router).

Volto is a game changer for Plone – you will immediately be impressed, we're certain of it!

Get your JSON Configuration In Line with GenSON

By Glenn Franxman ghostwritten by Aisling Fae January 29, 2021

There exists a great JSON tool called JSONSchema. It allows you to define a structure for all of your JSON configuration files to adhere to. This allows you to do things like implement JSON validation into your CI/CD pipeline — rejecting pull requests that try to merge invalid JSON.

But what do you do if you have years and years of JSON files piled up that were not made to adhere to a schema? Is there a way to bring all of these configurations in line? Yes, there is. It requires a bit of elbow grease and the use of a nifty little Python tool called GenSON.

```
$ genson *.json | jsonpp
{
  "$schema": "http://json-schema.org/schema#",
  "type": "object",
  "properties": {
    "Target": {
      "type": "string"
    },
    "TargetSchema": {
      "type": "string"
    },
    "TargetTable": {
      "type": "string"
    },
    "TableDefinition": {
      "type": "object",
      "properties": {
        "Distribution": {
          "type": "object",
          "properties": {
            "Type": {
              "type": "string"
            },
            "DistributionColumn": {
              "type": [
                "null",
                "string"
```

```
              ]
            }
          },
          "required": [
            "DistributionColumn",
            "Type"
          ]
        },
...
```
GenSON Example schema

Usually, you're using JSONSchema to validate new JSON files coming in. GenSON reverses this process by allowing you to plug in all of your JSON files. GenSON will then make a schema for you. It's not perfect; if you have thousands of lines of JSON scattered in dozens of files, a schema that validates all of them is going to incorporate all of the inconsistencies and off-by-one errors that have gone undetected.

What you can do then is fine tune it manually. You can use GenSON to spot the patterns — the bits of code that are consistent in the majority of your files — and you can decide to incorporate those into your schema and drop off the rest.

Say for instance you have to process through 5,780 JSON files, as I recently had to do. What you can do is write a schema that is hopeful — what you'd like all of your config files to adhere to (a sort of bare minimum). Then you validate all of your JSON against it, and you look at percentages.

If a deviation from your schema exists on more than some percent of the files, you can assume that one is actually intentional and can incorporate it into your Schema. If something is under that line you can investigate whether or not there is something that needs fixed manually.

It's a little time consuming but eventually you end up with a Schema you can use for any new changes in the future.

Here's a little Python script that demonstrates JSONSchema validation:

```
from jsonschema.exceptions import ValidationError, SchemaError
from jsonschema.validators import Draft7Validator
import json
import sys


python, schema_filename, *files = sys.argv


# the validator returns the reference to the rules and data
# as a Deq which doesn't have a useful __str__ or repr
# so we need a helper
def path2str(path):
    """ return a string for the json/schema path """
    pathstr = '.'.join(map(str, path))
    pathstr = f"@ {pathstr}" if pathstr else ""
    return pathstr


with open(schema_filename) as schema_file:
    schema = json.load(schema_file)
    validator = Draft7Validator(schema)

for filename in files:
        with open(filename) as config_file:
          config_instance = json.load(config_file)

        error_count = 0
        for error_count, error in enumerate(
                sorted(validator.iter_errors(config_instance), key=str), start=1
            ):
            print(
                f"{filename} - #{error_count} - {error.message}"
                f" {path2str(error.path)}\n"
                f"    see {path2str(error.absolute_schema_path)} rule"
                f" in {schema_filename}.\n"
            )

        if not error_count:
            print(f"{filename} validates.")
```

And here it is in action wherein I've intentionally misspelled a required JSON property in the file bad.json:

```
$ python ./validation_demo.py schemas/my_schema.json bad.json good.json
bad.json - #1 - 'TargetSchema' is a required property
    see @ required rule in my_schema.json.

good.json validates.
```

There's an old software engineering adage that says "minutes of testing save hours of coding." This applies here.

If you're just starting out a project that's going to make heavy use of JSON, you owe it to yourself to create a schema to handle all your JSON moving forward. Even if you already have a large number of JSON files with no schema, taking the hours to reverse engineer a schema for them will save you days of coding, reconfiguring, managing, and trouble shooting in the future.

Using this technique, I was able to generate a Schema file of 988 lines to manage all 5,780 JSON files. The whole process took about an hour and my client and I sleep easy, knowing that errors and inconsistencies can be caught quickly upstream and their configurations will remain consistent and reliable for years to come.