Python Flask vs Node.js

by Glenn Engstrand

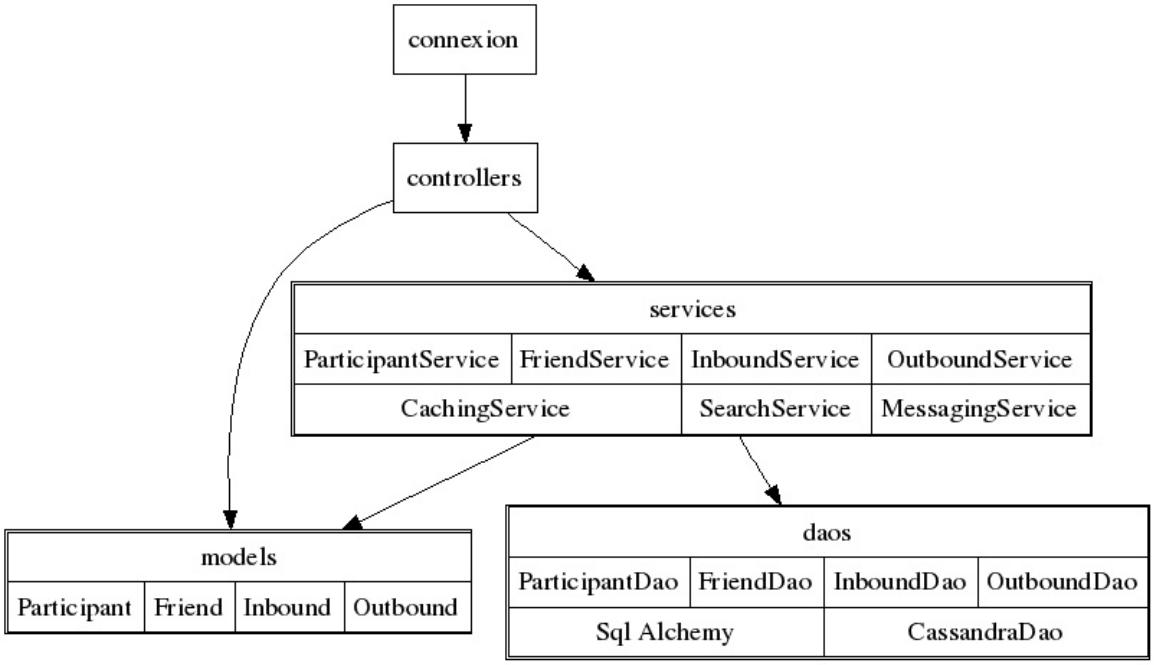
For those considering the development of micro-services, two very popular programming languages for that are Python and javascript and two popular frameworks for those languages are Flask and Node.js. You may be wondering which one to choose from. In this blog, I will explore what it is like to develop a rudimentary news feed micro-service written in version 3 of Python on Flask by comparing that experience with developing the same micro-service in Node.js

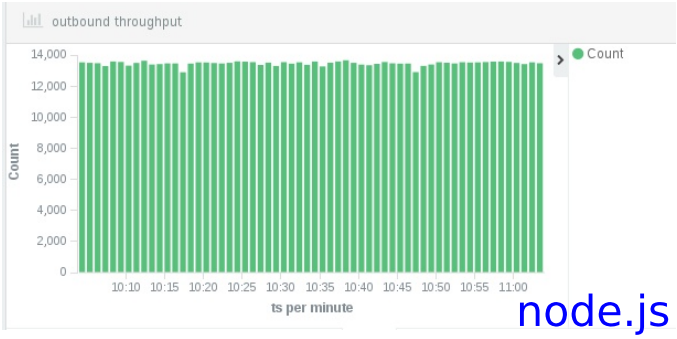
Tech Stack Rationale

Why choose Flask over Django? Isn't Django more popular than Flask? According to Google trends, it is three times more popular. According to Stack Overflow Trends, it is five times more popular. Django can be used to implement micro-services but it was originally intended to solve a different problem, namely how to develop a monolithic web site application. Why compare Python on Flask against Node.js? Many technology startups are disillusioned with Java and are seeking alternatives.

Python and javascript are the usual considerations. At first glance, they seem somewhat similar but there are some very profound differences too.

Both are Dynamic Scripting Languages which makes for a faster code, run, debug cycle. Neither programming language uses static type checking but both offer work-a-rounds for that. Both are enthusiastically embraced by the startup community, especially in the Bay Area of California. Both provide an integration with Swagger where the API specification itself is defined in a model contained in a YAML file.

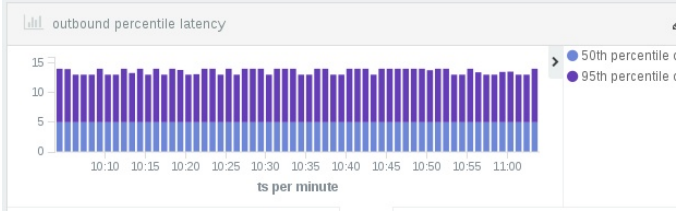




node.js

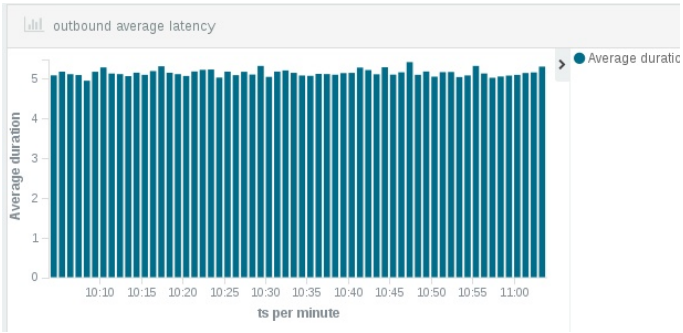
A build time generator uses templates to generate code and a runtime library is also used in such a way that the programmer doesn't have to write any code to implement the API itself. For Python, that integration is a project called connexion. For Node.js use the swagger-tools project.

Python 3 includes support for type hints baked into the language itself while javascript relies on jsdoc annotations and the closure compiler which is rarely used in server side development. Though still uncommon, you can write your Node.js apps in TypeScript which can perform static type checking. Node.js is more popular than Python Flask. According to Google Trends, Node.js interest is three times more than Python Flask interest. According to Stack Overflow Trends, Node.js interest is thirteen times more than Python Flask interest. According to Indeed.com, there are seven times as many Node.js jobs as there are Python Flask jobs with only a two percent reduction in average salary.



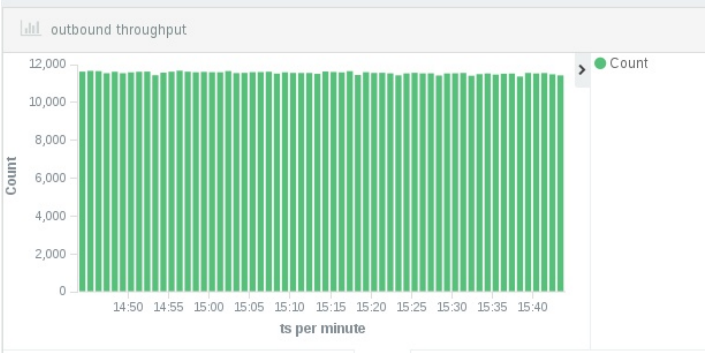
Architecture Differences

The most profound architectural difference between these two programming languages lies in their respective threading models. Theoretically, Python is multi-threaded but threads are under the control of the Global Interpreter Lock which means that at most only one thread can access any Python objects at any single time which means no true parallelism can occur in a single process. Javascript is also single-threaded per process and uses an event loop for its concurrency model. Another important architectural aspect in the two programming languages is how symbolic code gets converted into machine language. For Python, the source code is loaded and recompiled into the byte code if the modified timestamp of the former is later than the modified timestamp of the latter.

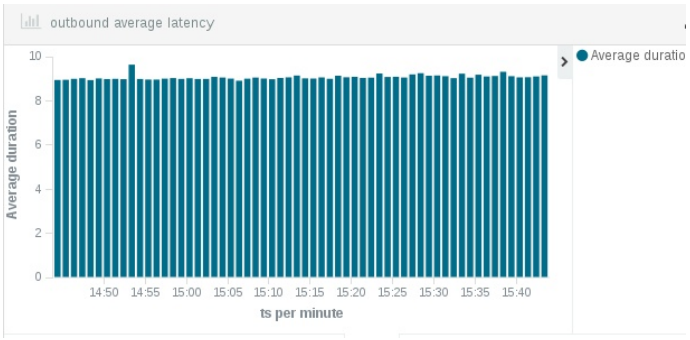


What both Flask and Node.js have in common is that they are minimalist web frameworks that are well suited for micro-service development. Flask does not concern itself with compilation but Node.js embeds the Chrome V8 compiler within it. This means that javascript code is compiled into machine language code in memory the first time it is referenced.

This is known as JIT or Just In Time compilation. After that, the machine language code is executed directly. Chrome V8 compiled machine language code executes faster than Python compiled byte code.



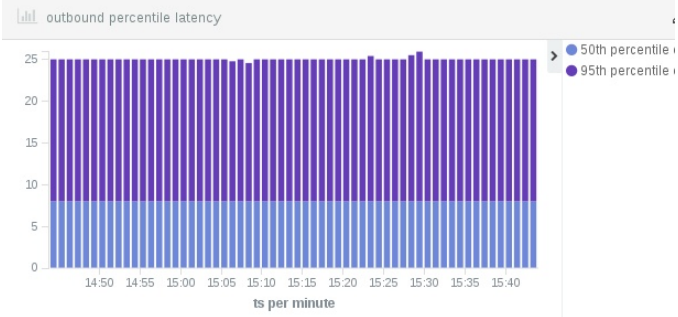
What both connexion and swagger-tools has in common is that they both read the same swagger YAML specification at the time that the application is launched and route requests based on that specification. How connexion differs from swagger-tools is that the HTTP server component is pluggable whereas swagger-tools just depends on Node.js which has embedded within it the HTTP server functionality. For connexion, the server backends available are tornado, gevent, and flask server. Though advise on the Internet recommends using gevent in production, my own tests revealed that the flask server had the best performance under load. Flask server can be configured to run in the following modes; single threaded single process, single threaded multi-process, or multi-threaded. Running load tests on all of these modes revealed multi-threaded to have the best performance.



Design Differences

Though javascript does support prototype based object orientation, development of Node.js services typically relies more on modules than objects. The opposite is true for Flask which depends heavily on object orientation. The service developed and tested here used the very popular Flask SQLAlchemy extension which provides access to MySql via an Object Relational Mapping layer.

Here is how the threading model differences manifest differences in how services are designed. For javascript, functions that depend (either directly or indirectly) on blocking I/O must use a callback mechanism where a function expression is passed in as an argument to the higher order function. Instead of returning the value from the blocking I/O, that function expression is called with the desired value. The lambda is also called when the I/O request failed.



Python / Flask

It is not correct to make any assumptions about the order in which these function expressions are executed in relation to code around it. Recent improvements include a promise based approach which is similar to callbacks but is easier to chain multiple blocking I/O calls together and separates the success processing code from the failure processing code. Python functions can safely return values from blocking I/O due to its threading model. Waiting for I/O to complete for one request does not prevent processing to occur for other requests.

	throughput (RPM)	average latency (ms)
tornado	3600	29
gevent	3670	33
multi-processor	769	134

Coding Differences

The Python Flask service has both twice the number of files and twice the number of lines of code than the Node.js service.

You would think that, since there is twice as much Python code as javascript code, that the Python code would be more complex. Actually, it is the opposite that is true. The Python code is heavily object oriented whereas the javascript code is not. The purpose of object orientation is to mitigate the complexity of the code. If you look at just the method that creates a new outbound post, then you will understand why the Node.js code is more complex than the Python Flask code. In both services, that method inserts a row into the outbound table.

The submitter's friends are fetched and corresponding rows are inserted into the inbound table for each friend. Finally, a document is inserted into the search index. In feed5 (Python Flask), that method is ten lines of code. In feed4 (Node.js), that method is 38 lines of code. The other factor that contributes to javascript complexity is managing all those call backs in a way that properly releases allocated resources both when the I/O succeeds and when it fails.

A very important requirement of unit tests is that they can be run separately from any dependent service. The tests should be able to pass even when the dependent services are unavailable or in a bad state. That is usually accomplished by employing a technique called mocking. For node.js, I used the Business Driven Development focused assertion library Chai and the mockery library for mocking modules that are responsible for accessing the dependent services. There is an extension to Flask called Flask-Testing which allows you to include coverage of the controllers in your unit tests but is not BDD focused. Because I was using Sql Alchemy for the relational database access,

I was able to reconfigure it to connect with SQLite instead of MySql for the unit tests. SQLite is a relational database for accessing a local file by a single user which is appropriate for unit tests. I used MagicMock for mocking the DAOs and service classes for the rest of the dependent services.

Community Differences

Both projects are hosted on GitHub so it is easy to compare them in terms of community focused metrics. Node.js is only one year older than Flask. There is easily an order of magnitude more commits to Node.js than to Flask. Both projects have lots of committers. Anecdotal evidence suggests that there is a lot more online content on Node.js than on Python Flask and that it is more accurate and helpful too.

The Python service can easily integrate with Kafka but I was never able to find a Node.js driver that worked. Since message brokers are pretty important these days and since Kafka is the leading message broker,

I would declare that this gap in the Node.js community makes it less stable than the Python community.

	node.js	python flask
Throughput (RPM)	13460	11557
average latency (ms)	5	9
95th percentile (ms)	9	17
CPU utilization (%)	72	47
network packets out (minute)	96000	31000
MySql connections	7	2

Both projects pride themselves on their bare bones nature. Instead of providing everything you need, these kind of projects focus on just being a micro-service container and let you pick and choose the other technologies that you will need. In this way, they are considered to be un-opinionated. Their main attraction to early stage startups is that the learning curve is very shallow which lets new programmers feel like they are being productive fairly quickly. Neither one of them has a lot of features. As stated earlier, the feed5 project here used Python version 3. Even though Python 3 has been around for over eight years, it is still considered to be immature. Its adoption has been slow due to its incompatibility with version 2. I believe Python 3 to be rock solid but this perception of immaturity becomes a self fulfilling prophecy. At the time of this writing, I had to run feed5 in docker because the standard EC2 AMI was incompatible with Python 3.

The tooling for both is fairly strong. When it comes to IDEs, pycharm is the most popular choice for Python and WebStorm is the most popular choice for Node.js development. I used emacs to code both projects. Package management for Node.js is npm and Python has pip. There are almost three times as many packages available in npm as in pip. I was able to get only one MySql driver to work with Python 3 and Flask SQLAlchemy and that driver cannot be installed by pip.

Performance Differences

Let's cover the most important differences which is in how well each technology performed under load on AWS.

In order to reduce the differences as much as possible, I commented out the calls to Kafka in the Python service and ran the Node.js service in a Docker container. Both services were tested in identical environments and with the same load test application.

The Python service was almost twice as slow as the Node.js service.

The Python service exhibited 14% less throughput than the Node.js service.

The Python service achieved a third less CPU utilization than the Node.js service. It also handled one third the network bandwidth.

Summary

The service written in Python underperformed when compared to the Node.js service. Not only was it slower, it also could not keep up in throughput and could not utilize as much of the hardware. I believe that the most significant reason for these differences lies in the difference in threading models. Python's threading model permits the Python coder to avoid the Node.js event loop inspired callback code complexity but, since only one thread runs at a time, no Python service can ever keep more than one CPU busy. Node.js is also single threaded but, with cluster mode enabled, the service can run single threaded, multi-process and utilize more of the server that way.

About Glenn Engstrand

Glenn's focus is on working with engineers in order to deliver scalable, server side, 12 factor compliant application architectures. He was a breakout speaker at Adobe's internal Advertising Cloud developer's conference in 2017 and at the 2012 Lucene Revolution conference in Boston. He specializes in breaking monolithic applications up into micro-services and in deep integration with Real-Time Communications infrastructure.