COMSE6156

Final Report

Phil Park

Pwp2106

**Monolithic vs Microservices Architecture**

1. **Introduction**

With the recent boom in web applications, new software engineering challenges have arisen. Companies want to provide service to more and more consumers and thus difficulties appear when trying to scale up their product. Many start-ups get off the ground running by implementing monolithic architectures, where many components and functionalities are tightly coupled together. The monolithic scheme has several advantages: It is often easier to implement; it is quicker to develop and get into a running state; it is relatively simple to add small additions to the product as it grows. However it is often not feasible as a long term solution. Many problems emerge as the developers want to expand their product; growing pains in a sense. For instance, it becomes more and more costly to scale up a monolithic application since you might need to scale only a certain functionality but are stuck with the overhead of the entire suite. An elegant solution to this problem is what is known as a microservices architecture. Netflix, Amazon and Uber are among the growing number of companies that are adopting their software architecture to the microservices model and have been seeing great success. Despite their glowing reviews, microservices are not perfect and possess many problems of their own. I analyzed and compared the merits and pitfalls of both philosophies, and would summarize them here in this report.

1. **Deliverables**

The application will be a simple message posting program. Users will be able to create accounts and post messages on a common board. The application will have user creation and deletion as well as message creation and deletion for each user. The app will also be executed via REST-ful API calls. The functionality of the program will be identical across both the monolithic and the microservices versions.

The final codebase along with installation instructions can be accessed here at <https://github.com/woodphil/columbia_micromono>

**2.1 Differences between versions:**

**2.1.1 Microservices Design:**

The microservices application split the user management/creation and the message creation and posting into two separate services. They were managed by a third controller application that will coordinate and control what the end user will see. The controller will be responsible for sorting out which requests and responses go where.

User Creation and Management

Controller

Jinja Template Frontend View

Message Creation and Management

The two modules have their own sqlite database to connect to and read write from.

**2.1.2 Monolithic Design:**

The monolithic version will be a standalone application. It will handle all the business logic and rendering in a Model View Controller architecture.

Jinja Template

(VIEW)

Flask Application

(CONTROLLER)

SQL Database

Model Definitions

(MODEL)

1. **Results:**

**3.1 General Differences:**

Overall the development time taken for the microservices version of the application took longer. I documented 10 hours to complete the monolithic version and 14.6 hours to complete the microservices version. The total lines of code written for the monolithic version was also less than the ones written for the microservices version, 151 vs 240 respectively. This was not too unexpected however, since I had to essentially write three mini applications, there would be triple the boilerplate code for running each application. The development process for the monolithic version was fairly smooth and straightforward as expected. Creating a simple application was relatively simple in a monolithic app. Conversely, when I tried to design and implement the same functionality in a microservices architecture, several challenges were encountered.

**3.2 Challenges:**

There were 3 main challenges that were uncovered while building the project.

The first was the change in how data needed to be processed. Communication of data between modules could no longer be within an application. That means native python data could not be easily passed between the user and message controllers with the actual site controller. Instead of passing the data directly as an argument to a function within the controller, I encoded the data into JSON format using packages to pass them via REST calls. This added a lot of work since now every function call and response that crossed applications required its data to be encoded in some format that ideally had to be consistent over all of the microservices. A publication/subscription (PUBSUB) model is a fairly popular system used for large amounts of microservices. It involves individual microservices publishing events upon doing/receiving events and other microservices which are subscribed to said events that are able to be notified when these they are published. Additional components such as load balancers and work queues should also be added to help optimize job distribution across multiple instances of microservices. After all, the whole point of microservices is to duplicate many instances of processes in order to scale efficiently.

The second challenge was now data storage. Since microservices are aimed at optimizing scalability, having a single database system for the entire application is a no go. Unfortunately I was not able to implement an optimal solution in my application for this problem. I wanted the message module and the user module to have their own separate database system, but I discovered that by doing so you lose important features such as foreign keys. However despite these difficulties, spreading out databases is definitely the solution for a microservices architecture. Isolating each database now allows each service to be relatively unaffected by changes in other databases, it also lets each service choose a database system that is optimized towards their specific task. A proposed solution to resolve the loss of connectivity between services was to have the join occur in the separate web application controller rather than the database itself by querying the separate services and gathering and processing their results together. To me that was a very foreign concept, having worked with relational databases mostly. I would have liked to have implemented such a system but it was not possible in the time available.

The final challenge was the debugging process for microservices. Since the web application controller, user controller and message controller were three separate applications, debugging became very annoying very quickly. When a service failed, the default Flask error output was generally unhelpful at quickly pinpointing the source of error. To overcome on my end I encoded the error outputs of each application into a JSON object that would be trickle up to the main application. By doing so I did not have to monitor and dig through several applications to identify the sources of bugs. While this solution works on a small scale, better systems need to be in place when an application is scaled up to dozens to hundreds of instances spread across multiple machines. One solution is to use distributed debuggers which are designed to monitor multiple processes simultaneously for error catching. Another solution is to give every microservice the ability to publish an error event and have a single error consuming microservice that handles error reporting from this queue.

1. **Conclusions**

I found the final project to be a very good learning experience. In order to really execute a good microservices architecture, the design has to be very strict. I see why it is less popular than monolithic applications due to its rigidity, and synchronization of many components. My own results found that I spent significantly more time designing and implementing the microservices than the monolithic design. While developing the monolithic application, things were definitely faster and looser and it was easy for me to implement the necessary features. With the microservices, it took me much longer and I stumbled quite a bit more. However I do think the end result is much more robust and modular. As I discussed in my paper I still believe that the monolithic approach is better for starting up your ideas and getting them out there. The microservices design is much more detailed and requires a lot more work to get the systems running. I do believe in most cases it should be adopted when scalability has become a noticeable issue, later on in an application’s life.

1. **Resources:**

Flask: http://flask.pocoo.org/

Flask-RESTful: http://flask-restful-cn.readthedocs.io/en/0.3.4/

Flask-SQLAlchemy: http://flask-sqlalchemy.pocoo.org/2.1/

requests: http://docs.python-requests.org/en/master/

simplejson: http://simplejson.readthedocs.io/en/latest/