Tech Review: Server Application

Henry Peterson, Server Architect

October 11 Fall 2018

CS461 Senior Capstone, Group 19

Abstract

This paper details the options for technologies used in our server application. From the available choices, I recommend Node.js for our implementation framework, a micro-service design for our architecture, and Jest for our testing framework. I selected Node.js because it will provide the best time for development given the size of the project. I selected a micro-service design because it best suits our needs and is already laid out for us. I selected Jest for the testing framework because it is the most robust that can provide quick results.

CONTENTS

I. Language

Language choice has a significant impact on the structure of an application. It is important to pick the one that will best suite your needs so that it will make your work more efficient instead of hindering it.

A. Node.js

Node.js [1] is a frequently used technology, allowing for servers to be written in JavaScript, which is traditionally used on the front end. One of the most convincing reasons to use Node.js is that we have an existing server written with it. Improving code takes much less time than creating it and there is no reason to throw out work that has already been done. It is under the MIT license which means we can freely use and modify it, but we do not have to publish our use of it in a proprietary system [2]. An issue with our current implementation using Node.js is the lack of type safety. While this can make development time faster, it can reduce code clarity for teammates or future developers as well as make it harder to ensure code correctness. We would want to correct this by converting the code base to TypeScript, a statically typed version of JavaScript [3].

B. Ruby

Ruby is also popular with its implementation, Ruby on Rails. It is another serious contender as we also have a version of the server written with it. It has an emphasis on an Model View Controller (MVC) pattern, which lends itself nicely to how our application is structured and is also under the MIT license. The MVC pattern which would represent our front end as the views, the server as the controller, and our database structure as the model. Although it has its qualities, this implementation was discontinued by the last group in favor of the Node.js version. We would have to put in a significant amount of effort to get it to a comparable state. In addition, it lacks type safety, but there is no strong type-safe equivalent, so a conversion is not possible.

C. .Net Core

.NET Core is yet another framework by Microsoft available for most system, distributed under the MIT license. It uses C#, which is type safe, and emphasis the MVC pattern. Our team members are most familiar with languages in the C family or even specifically with C#, so writing an application using it would be much smoother. There is also an argument that a C# implementation would be more performant, but with our use case, that is not a primary concern [4]. The main downside to it is that there is absolutely no existing code in the project in .NET Core so we would have to discard perfectly functioning code. There is not much difference in what it provides compared to the others; they mostly differ in how their goals are accomplished.

Of these three options, my recommendation is Node.js. .Net Core has speed and Ruby has the focused design strategy, but speed is not a primary concern and our design fits it Node.js. While converting to TypeScript will take some effort, it is far less work than fixing broken code or creating entirely new code. In addition, it will improve the robustness of the code, so it will not be worthless effort.

II. Architecture

The architecture of a server dictates how interactions with other computers are modeled. Our system needs to be able to interact with just a few devices making requests at a time and interact with a mid-sized database.

A. Micro-service

A micro-service architecture involves breaking down the components into smaller, function-specific programs, or ”services”. This allows development and maintenance to be done in more bite sized chunks. When changes need to be made in one area, there is less concern about affecting an unrelated section, because they are separate. As long as the interface between them is consistent, error are less likely [5]. Both the existing Node.js server and Rails server are built using this model. Our project is relatively small so it can be broken into two manageable components pretty easily: the server and the database. Currently, Docker is used to run these pieces separately.

B. Event-Driven

An event-driven architecture is used in an effort to use a server’s resources more efficiently. Instead of communicating with a client for the entire exchange, it can take information and give updates only when changes happen. This allows event-driven servers to be much more scale-able than a typical micro-service architecture. But, aside from the time that it would require to redevelop an application with this model, it is not necessary for our needs. We have a small and static number of users who will be connecting and known amounts of data being exchanged, so scaling is not a concern.

C. Layered

A layered architecture is similar to a micro-service architecture, but it has physical separation of its components. They are separated in a way similar to the micro-service would, but each component having its own machine has added power and security. Each piece has its own processor and memory to utilize. If one section goes down the others are still up to be able to prevent data loss [6]. While these are nice features, it can also be very expensive. The machines alone cost a lot of money and the added maintenance and overheads adds even more. This can be outweighed by the benefits is a large scale operation, but again, with our size, it would be overkill and end up costing more than is gained.

Of these three options, my recommendation is the micro-service architecture. Its features suit our needs as well it being the architecture of choice in the current implementation. Reimplementing the system for scalibility is not something that would be in the scope of this project.

III. Testing

Code testing is a part of a project that often does not get recognized by users, but is a crucial piece in having an efficient development process. Testing frameworks are generally made for a specific programming language, so the ones covered here are for Node.js, the recommended language option.

A. TSunit

TSunit is a testing framework specifically for TypeScript. It focuses on providing robust testing facilities in the same language it is written for [7].Many testing suites are written for JavaScript then adapted to use with TypeScript. Avoiding this allows us to avoid additional dependencies and complexity. However, TSunit is not a very mature library with little adoption, so we may be prone to running into significant bugs that will not be fixed in a timely manner. This project is on a rigid time schedule so selecting a robust option that allows for efficient work is essential. Further, conversion to TypeScript is not set in stone, so going for something that is so tied to it may lead to more work down the road.

B. Tape

Tape is a testing framework that focuses on having minimal complexity, but at the cost of less features [8]. Our project has a relatively small scale, so minimal complexity is preferable. We will only need to implement basic testing, which means having a quick set up time for the framework will yield the best results. On the other hand, it does not integrate with typescript will, so it may lead us to either using JavaScript, or putting more effort into making it function with TypeScript than is worth.

C. Jest

Jest is a feature rich testing framework that has built-in Typescript support. It does come with some overhead such as potentially polluting the global namespace, meaning that it is not very modular and it can possibly conflict with other names used in the program. But this is a negligible issue in a small scale project. A strong point for Jest is that it is well established and documented [9]. Complexity is easily combated if there are ample resources to find answers to questions, which Jest does provide. In addition, some of our team already has some experience with Jest, so overhead can be minimized even more.

Of there three options, my recommendation is using Jest. While it may be a little large for our needs, it will work well and with our existing knowledge it can be used quickly. Any additional resources used will not be in the final product when it runs either, only on the developer's side, so it should not have a large impact on the client either.

References

[1] N. Foundation. https://nodejs.org/en/about/, Novermber 2018. (Accessed on 11/02/2018).

[2] https://opensource.org/about, Novermber 2018. (Accessed on 11/02/2018).

[3] J. Nance. https://stackify.com/typescript-vs-javascript-migrate/, September 2017. (Accessed on 11/02/2018).

[4] J. Daniel. https://raygun.com/blog/dotnet-vs-nodejs/, May 2017. (Accessed on 11/07/2018).

[5] S. Bear. https://smartbear.com/learn/api-design/what-are-microservices/, November 2018. (Accessed on 11/02/2018).

[6] A. O. Ramirez. https://www.linuxjournal.com/article/3508, July 2000. (Accessed on 11/02/2018).

[7] S. Fenton. https://github.com/Steve-Fenton/tsUnit, July 2018. (Accessed on 11/02/2018).

[8] Ben. https://raygun.com/blog/javascript-unit-testing-frameworks/, May 2017. (Accessed on 11/02/2018).

[9] Jest. https://jestjs.io/docs/en/setup-teardown, November 2017. (Accessed on 11/02/2018).