Easing the facilitation and creation of scalable global internet based applications

Omar Elamri

The purpose of this study is to help developers rapidly create scalable applications on the global scale. Current research shows that today's methods are slow and inefficient—especially for mobile devices which are on the rise. If developers want their applications to be accessible to more users, current methods will have to change. This project streamlines the development of applications—in essence creating a one stop solution for all a developer's needs. Both the backend, frontend and database solutions are abstracted to a singular framework. This is achieved by creating a unified framework, load balancing databases and servers, and finally using serialization methods that allow for faster and more efficient data transfer.

Keywords: *backend, frontend, React, NoSQL, nonrelational, sharding, load-balancing*

Mobile devices are becoming more abundant which in turn increases the reliance on cellular networks (that have less reliable connectivity than broadband). In order to serve these new needs, internet-based applications need to be well-versed in complicated compression, parallel processing, and sharding of databases over a global network (Mishra, 2020).

Applications must be able to interact with both their backend server and frontend application with ease. Serialization methods that efficiently transport data between the two all have their drawbacks. In addition, the new social media-based age requires all data to be transferred in real-time. Transmitting data from different corners of the world in real-time is a challenge. Also, developing requests and responses and processing them on both sides can be difficult at times. A unified programming approach for both the frontend and backend is easier than current solutions and will be utilized to aid the developer's linkage between the two.

This research seeks to create a way for application developers to more easily create applications that meet the new standards required by less stable connections. It will be a unified approach to tackle all these problems in a single framework. Unlike the other frameworks, this one will not merely focus on either the backend or frontend, but both. It will also support running on an independent infrastructure or a current cloud virtual machine provider. The research dedicated to developing this new framework could potentially have major implications. For starters, it will make application development easier, so there will be an influx of indie app developers who no longer need to worry about the specifics and be free to unleash their creativity. In turn, this could make the internet more accessible for all and reduce society's reliance on megacorporations for our internet access.

The focus of this study is to determine how to best serve the needs of up and coming software developers so that they can build reliable, global-scale, and real-time applications. In addition, the study will delve into developing such a framework so that the above is possible and easily facilitated. Applications are becoming harder to develop day by day—and with each of the new standards and demands of the modern world—it's becoming increasingly difficult for developers to keep up. The results of this study and its framework could potentially streamline application development for all—democratizing and demystifying the process while at the same time providing more of a choice for the world to make a greater number of applications by newer developers.

Methods

The cornerstone of this project, the unified framework, was created by providing a wrapper around two individual frameworks. For the backend, it uses a custom WebSocket handler previously created by the lead researcher. The frontend uses a React sample project. The framework listens to backend calls such as database requests and updates, whereas the view logic is implemented on the React side. Additionally, the framework provides the ability for the developer to designate exactly which code runs locally or on the server. The combination is implemented by starting out from a React project. The framework seeks out the individual snippets and compiles them into a backend executable.

Secondly, a real-time database that is non-schema based (NoSQL) was created. This database provides structure alongside a sharding system that allows for real-time communication between shards. (Note: NoSQL databases typically imply a key-value pairing system.) Current implementations such as MongoDB exist; however, they are too large (and made for commercial use) to be viable for this project. Thus, it's imperative to create a new database. Databases need to follow the ACID guidelines—in other words, databases need to be atomic, consistent, isolated, and durable. Atomicity requires that a single transaction either results in a change or not at all. If the connection is lost in the middle of a delete transaction, it could result in a partial deletion. This makes the record in a state of limbo. Isolation requires that one database transaction results in one change. This may seem simple, but an update transaction consists of two individual transactions: an addition, and a subsequent deletion. If two update transactions occur at the same time, the ordering of the four transactions could be out of order and may result in a net deletion—not isolated at all. In addition, a strong sharding system for the databases was implemented. In a global application, data is shared across the internet—it's infeasible to have a copy of all the data on each database; therefore, data needs to be spread across the shards. This project created an algorithm that efficiently allocates data between shards. Also, the communication between the shards needs to be secure. For the interim, a simple HTTPS over TLS protocol suffices for this communication; however, it's uncertain whether or not this is fast enough for the vast loads of data.

Results

The unified framework was developed in actuality by simply creating a new file in the frontend source code. The file could have any name, but as a default, it creates *catalyst.js* or *catalyst.ts* in a TypeScript project. In this file, the developer writes functions that can be referenced in the frontend code as seen in Fig. 1 and Fig. 2.

```
unifiedapproach.testfunction =
(param1, param2) => {
console.log("testfunction")
}
```

Figure 1. Function creation for use in the frontend

```
<p
onclick={unifiedapproach.testfunctio
n(1, 2)}></p>
```

Figure 2. Referencing function in frontend code

On compile time, the framework splits the two into its constituent parts: the frontend and the backend. It changes *package.json* to run the framework's executable when running npm's build script. Firstly, in a TypeScript project, the framework tells tsc (the TypeScript compiler) to ignore *catalyst.ts*. (In a JavaScript project, the framework simply moves the file to another directory.) Secondly, a "new" *catalyst.ts* is created, but only with functions that connect to the WebSocket connector. This is done by extracting the functions' parameter and return type using regular expressions (Fig. 3).

```
var STRIP_COMMENTS =
/((\/\/.*$)|(\/\*[\s\S]*?\*\/))/mg;
var ARGUMENT_NAMES = /([^\s,]+)/g;
function getParamNames(func) { var
fnStr =
func.toString().replace(STRIP_COMMEN
TS, ''); var result =
fnStr.slice(fnStr.indexOf('(')+1,
fnStr.indexOf(')')).match(ARGUMENT_N
AMES); if(result === null) result =
[]; return result; }
```

Figure 3. Extraction of argument names (Allen, 2020)

This new *catalyst.ts* will be compiled alongside the rest of the frontend in addition to boilerplate code that facilitates the WebSocket connection with the backend. Thirdly, the original *catalyst.ts* is transferred/copied to another directory which will serve as the backend of the application. Boilerplate code such as creating the WebSocket listener is added, and a working backend is created. From a single codebase, both the frontend and backend are created.

The real-time database commits changes to a ledger and synchronously commits them to each constituent document file (Fig. 4). Firstly, a change is made by calling any of the built-in document altering functions. Once they're called, the changes will be serialized into a format that fits into an array—which is stored in memory by *ledger.ts*. Before the change is added to an array, *ledger.ts* checks for any conflicts with previously committed changes. If there is a conflict, it throws an error to be handled by the caller of the altering functions. This array acts as a queue—where *ledger.ts* incrementally dequeues each change onto a file (*ledger.txt*). The usage of an intermediate file is to ensure changes are committed (once the executable goes back online) even when execution is halted. The main database engine then reads from *ledger.txt* and makes the changes synchronously to the database. This ensures that the ACID guidelines are followed. Document altering methods are available, but in the interim are limited to modifying document properties. These include *insert* and *edit*. In addition, complex data structures such as arrays are not supported natively (however, a client can easily serialize one using a string).

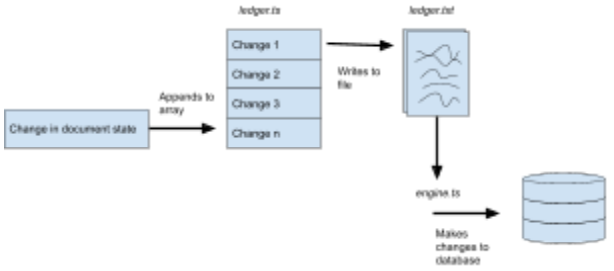

Figure 4. Shows the process in which a change is made on the database.

Discussion

The unified framework, while useful, does have its limitations. Firstly, external libraries are sparingly supported. This is due to the fact that code is simply copied from one file to another. Unlike the code itself, dependencies aren't added to the newly created backend. This could be mitigated by extracting which libraries are used from the import declarations in addition to the dependencies stated in the frontend's *package.json*. This would be the primary facet of a future project due to how important it is for libraries to be supported—this framework would be functionally useless without them. Another limitation is type checking. While *catalyst* supports TypeScript, it does not inference return types and parameter types. This could be mitigated by either using a similar regex approach to the extraction of argument names (seen in Fig. 3), or attaching to the TypeScript engine. The final limitation is the lack of automated deployment. It's currently up to the developer to create the SSL certificates for both the production and development servers along with the configuration of how the server is handled by the operating system. For the latter part of this limitation, pm2 (keymetrics.io, 2014) is a suggested library to run the backend server. This method works for developers running on their own hardware, but it doesn't provide tight integration with cloud platforms such as GCP/AWS. These platforms have built in load balancing that can be utilized.

For the database, the clear limitation is the lack of document altering methods. While this database has two crucial methods, MongoDB has over thirty (Mongo, 2021). However, this is also a non limitation as the lack of methods lessens the resources required to run the database—making it more efficient. Arguably, this efficiency is minimal compared to the infeasibility of using a database system with so few methods; hence, a future project will have to create more.

All in all, while this project isn't realistic for actual usage, it serves as a case of proof for the feasibility of a future project. A unified framework and tightly integrated database can work in conjunction with each other, but more development and feature compatibility is needed for it to be viable.

References

Mishra, S. K., Sahoo, B., & Parida, P. P. (2020). Load balancing in cloud computing: A big picture. Journal of King Saud University - Computer and Information Sciences, 32(2), 149-158. doi:10.1016/j.jksuci.2018.01.003

Allen, J. (2012, March 29). How to get function parameter names/values dynamically? StackOverflow. Retrieved May 10, 2021, from https://stackoverflow.com/questions/1007981/how-to-get-function-parameter-names-values-dynamically

keymetrics.io. (2014). ADVANCED, PRODUCTION PROCESS MANAGER FOR NODE.JS. PM2 - Home. Retrieved May 17, 2021, from https://pm2.keymetrics.io.

“Mongo Shell Methods” Mongo Shell Methods - MongoDB Manual, docs.mongodb.com/manual/reference/method/.