One Solution of Component Based Development in NodeJS for Modularization of gRPC Services and Rapid Prototyping

Mihailo Vasiljević [1], Aleksandar Manasijević [1], Aleksandar Kupusinac [1], Ćamil Sukić [2], Dragan Ivetić [1]

[1]*Faculty of Technical Sciences University of Novi Sad, Novi Sad, Serbia*
[2]*Department for Computer Sciences, University of Novi Pazar, Novi Pazar, Serbia*

Abstract **– Developing software system as a composition of components that are independently created, provided and installed can significantly improve the value of software solution. In this paper we offer one possible solution for creating gRPC services, using NodeJS, as independent modules, components. We showed all main structures and sketched all main components.**

Keywords **– grpc services, microservices, independent modules, modularization, nodejs.**

1. Introduction

Trying to tame complexity of the software is something that engineers have been attempting to achieve since its beginning. In the modern software architectures there is a trend of using microservices to fight against monolithic structure of software. Microservices are characterized by organization around business capability, automated deployment, intelligence in the endpoints and decentralized control of languages and data. [1] The idea of developing software as a composition of independent parts is in the roots of component-based software engineering. This paper is describing how combining those two, in the foundations very similar, ideas can lead to developing loosely coupled software systems with high focus on reusability, scalability, and extendability regarding the parts of the system or of the system as a whole.

DOI: 10.18421/SAR24-06
https://dx.doi.org/10.18421/SAR24-06

Corresponding author: Mihailo Vasiljević,
Faculty of Technical Sciences University of Novi Sad
Email: mihailov@uns.ac.rs

Received: 05 November 2019.
Revised: 11 December 2019.
Accepted: 18 December 2019.
Published: 30 December 2019.

2. The method

Node.js is a free server environment (open-source) written in JavaScript. It can be run on various platforms (Windows, Linux, Unix, Mac OS X, etc.). Node.js provides many libraries that can be used in order to develop web applications more easily. [2][6]

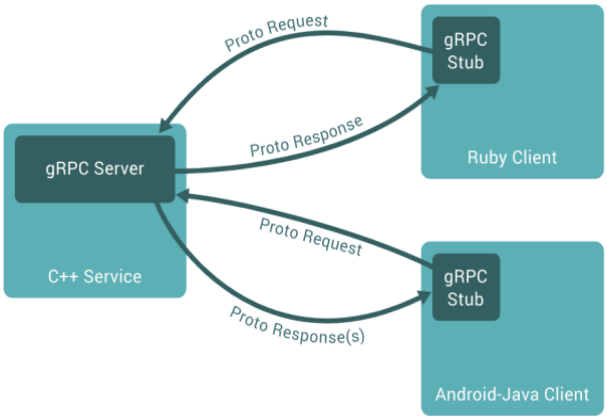

Figure 1. Ilustration of gPRC concepte

Using gRPC, it is easier to create distributed applications and services. This is because the client application can directly call methods on a server application that is on a different machine as if it was a local object. The main idea while using gRPC is defining a service with methods that can be called remotely with specific parameters and their returned type. So, gRPC server implements this interface and it is used for handling client calls (calling previously specified methods). On the client side, there are same methods (stubs) provided as on the server. One of benefits of using gRPC is that it is not important what environments you are using on server side and what environments you are using on client side. You can for instance create gRPC server that runs Java code, and client app that is written in Go, Ruby etc. [3]

Component-based development concept is very common concept in software development. It is very widely used. The main thing of this concept is reusability of previously created (already existing) components and adding new ones in the existing architecture. Simple object accesses protocol (SOAP), ActiveX and remote method invocation (RMI), which are ones of the well-known architectures of component-based development. One of the most important and one of the major advantages of developing software applications using component-based development concept is reusability. There are few more advantages when using this concept like upgradability, interoperability, less complexity, reliability, improved quality, cost effective, efficiency etc. [4],[5].

Object Oriented Methodology is an approach where data and functions are integrated. Data is stored in objects which can be replaced, modified and reused, and therefore, software is a collection of this objects that are representing model from real life. Class in Obejct Oriented Methodology is a group of objects that share a structure and that have similar operations. This concept is very different from the concept of structural programming because we can get higher productivity, lower maintenance cost and get better software quality [8],[9].

3. The result (Separation of concerns for gRPC services in NodeJS)

3.1. Platform package diagram

The application itself has an extremely simple package structure, since it is only a foundation on top of which other applications should be built. The structure can be seen in Figure 2. In the center of the structure is a platform package that uses shared functionalities accumulated in common package. Since every operation of the platform should provide executions inside transactions, in order if something happened to be easily reverted, the transaction package provides command pattern implementation. gRPC Server startup is done in index.js file that can be found inside the app package. Since one complete component-based system needs some user interface in order to provide interaction with its platform there is platformClient component.

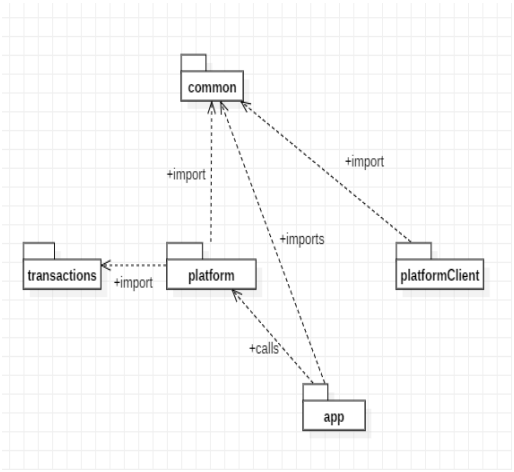

Figure 2. Package diagram of Nodejs gRPC CBD platform

The methods offered by this service correspond to those of the platform. This makes it possible to add gRPC services without restarting the gRPC server. In this way, each service can take care of its own dependencies, and fully develop as a true separate component.

3.2. Common package UML diagram

Common package contains all functionalities that should be shared between other packages. The main focus is on abstract service class. UML diagram of common package can be seen in Figure 3.

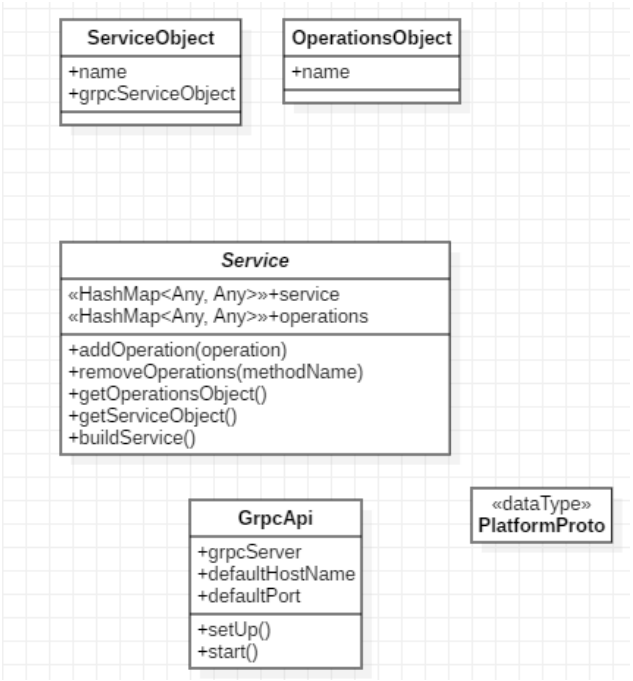

Figure 3. Common package UML diagram

Since JavaScript does not provide a mechanism of defining abstract classes, the regular class is used for defining service abstraction (referred to as "abstract service" in further reading), using the well-established convention that all abstract methods that are not implemented should throw an appropriate exception.

gRPC server expects two parameters for his addService() method, and those two are:

1. gRPC service object
2. Object that contains methods gRPC service provides

In order to be compatible with this gRPC server request, abstract service provides two methods for retrieving those two objects from the concrete service. The model is simple, service has a unique name, a property which key is provided unique name and value gRPC service object, and operations object which is populated by concrete service's methods. Abstract service provides helper methods to classes that inherit it to populate or remove methods from operations object. One more method is provided. That is the one for building service. In essence, in that operation, methods of concrete service should populate adequate operations object. Build service method is abstract by its implementation so, as it was already mentioned, it should be overridden in classes that inherit abstract service.

This abstract service is in the main focus since it provides necessary and sufficient amount of operations for declaring gRPC service.

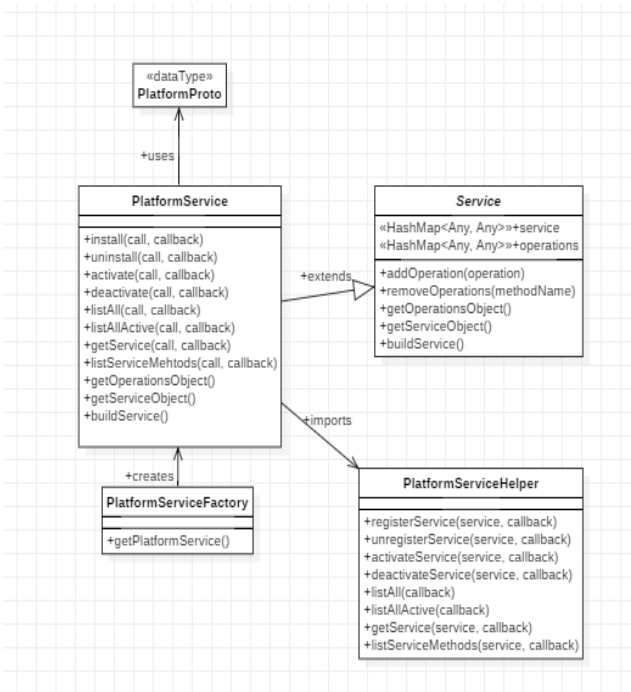

Figure 4. Platform package UML diagram

There is also GrpcApi class which is singleton and should be used for making instance of an actual app object. It contains gRPC server object, host name and port on which app will listen for requests. All the setup is done in setup() method, and app itself can be started using start() method. The intention is to completely isolate and localize the process of creating an application object.

Platform protocol buffer is represented as data type on this UML diagram.

3.3. Platform package UML diagram

UML diagram is represented in Figure 4.

Concrete services can be made by inheriting abstract service. They are only obligated to provide the methods defined in service protocol buffer, and as we earlier mentioned, the method for building service. In order to band together concrete service's methods with the operations object in abstract service, provided methods of super class can be used.

In order to tie up everything, a well-known pattern for simple inversion of control had been used.

Firstly the method of extending gRPC server object used is delegation. Appropriate wrapper around gRPC server had been created. The wrapper itself reference factory method for creating an object in charge of dynamically requiring concrete services. The list of services should be provided to the factory method itself in the form of an array of services' relative paths. After service is loaded it is kept in memory for further usage. Consumption of service means using provided abstract service methods for object building and retrieval.

The developed platform relies heavily on a modular development system built into NodeJS itself. The code is grouped into packages, so-called npm modules and distributes via node package manager (npm). Some bare minimum that the platform of this kind should provide is the ability to install and uninstall services, their activation and deactivation, the ability to list active or installed services, as well as listing methods that a service provides to other services connected to the platform. The existence of such a platform is not the standard for NodeJS projects, so there are already such, mostly in-house solutions, which are either tied to a specific project or extremely complicated for rapid prototype development. For this reason, the platform in question offers a core set of operations with the main focus on extremely small learning curve, rapid adoption into existing projects and even faster

implementation on new projects. It is recommended for use on small projects or prototypes [7].

The platform itself is an npm module, so it is distributed in the same way as its parts. It is in the form of a library, so it is independent of specific projects. It offers the necessary minimum of operations previously described.

The main object is PlatformService which is also refered as context. As can be seen in the UML diagram of Figure 4, the context itself, within its state, stores information about the services to be loaded and the service objects that are loaded. Installation/uninstallation and activation/deactivation operations are performed asynchronously since they can take a long time if the dependency graph of the module being installed is large.

The operations for listing all available modules, listing all active modules, supplying a service object, and listing all methods of a service are not implemented asynchronously since they are not time consuming. In case they would need to be executed asynchronously as well, since functional programming was used for their implementation, switching to non-blocking mode would be as simple as direct use of javascript key words async/await.
It is important for the platform to remember the activated modules between the two starts. This is implemented like the inversion of control specified in section C. After each activation and deactivation, the corresponding file containing the array of active components is updated by adding or removing the component from the array of active components. Given the main idea that pervades the entire platform, which is its simplicity, the use of more complex storage mechanisms is not necessary, thus it is avoided.

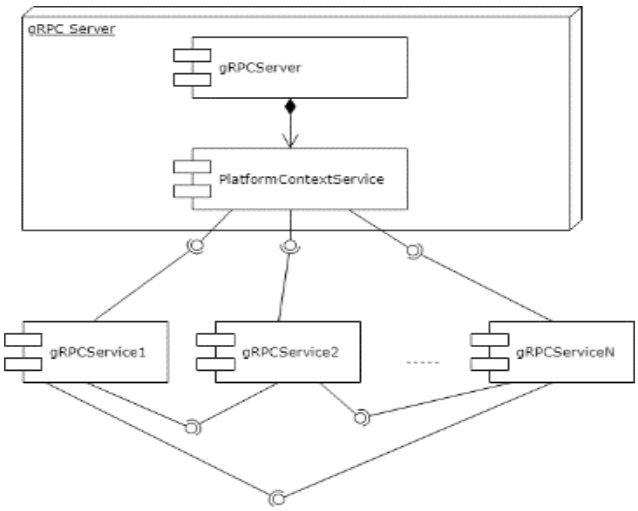

Figure 5. Component model diagram of gRPC modular architecture

As the component model is not strict, the components are not required to implement any interface. The platform can be used and customized for the prototype of an application based on the components with a potential 100% reusability without modification.

Combining the elements described above can affect the flexibility and scalability of gRPC NodeJS applications. A solution that integrates a component-based development platform and offers a separation of gRPC concepts implies that gRPC services are written as separate components, npm modules, registered and activated on the platform, and as such available for use, as shown in Figure 5.

Integration of a component-based development platform involves the formation of a gRPC service in the manner described in segments B and C, whereby the methods offered by this service correspond to those of the platform. This makes it possible to add gRPC services without restarting the gRPC server. In this way, each service can take care of its own dependencies, and fully develop as a true separate component.

3.4. Project structure

Organization is the key concept for speed and, in the end, quality of software development. Typical project structure for one NodeJS and gRPC project can be the one shown in Figure 6.

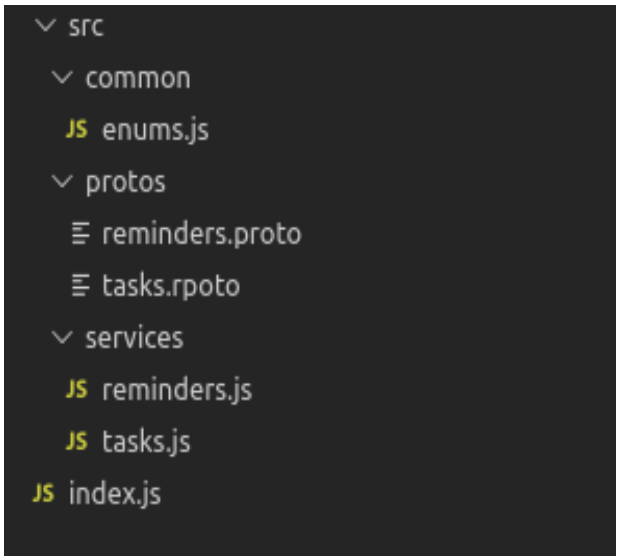

Figure 6. Example of a horizontal gRPC project structure

This project structure is usually acceptable for smaller projects, but the extension of the project and the constant addition of new files can lead to clutter in which it is difficult to navigate. Even though the solution proposed can easily be applied using this

kind of project structure, in the original implementation, and having in mind component-based development, project structure is used, and it is called vertical, or feature-based project structure shown in Figure 7.

As it had been shown earlier, extracting features in separate modules can easily be done by using the recommended project structure, or they can be easily linked as submodules from version control systems without making a significant impact on future development process. One more advantage could be that code generators can more easily create new features.

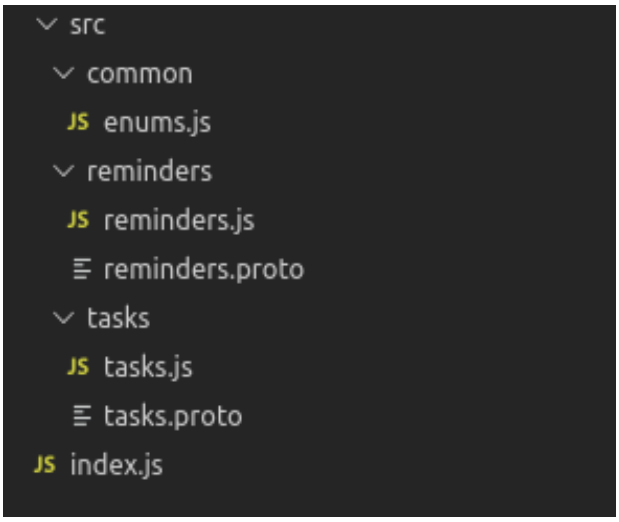

Figure 7. Example of a vertical, feature-based gRPC project structure

4. Conclusion

The architecture of gRPC NodeJS applications can significantly affect the development process. Quality architectures are easier to understand, new members become more involved in the project, since they can focus on getting to know the individual feature they need to work on, rather than the whole project. Separating the responsibilities of the concepts used, results in easier extendibility, scalability and overall flexibility of the solution. The ability to quickly form an application prototype has an extremely positive impact on agile development. A simple library that does not require a steep learning curve can have a positive impact on the formation of rapid prototypes and, subsequently, an extremely simple mapping to more serious industry-standard solutions.

Acknowledgements

This work was partially supported by the Ministry of Education, Science and Technological Development of the Republic of Serbia within the projects: ON 174026, III 42004 and TR 32044.

References

[1]. Martin Folwer, "Microservices". Retrieved from: https://martinfowler.com/articles/microservices.html. [Accessed: 22 August 2019].

[2]. Tutorials Point, "Node.js - Introduction, Tutorialspoint". Retrieved from: http://www.tutorialspoint.com/nodejs/nodejs_quick_guide.htm [Accessed: 05 September 2019].

[3]. Google, "Documentation". Retrieved from: https://grpc.io/docs/guides/ [Accessed: 05 Sep 2019].

[4]. Qureshi, M. R. J., & Hussain, S. A. (2008). A reusable software component-based development process model. *Advances in engineering software*, *39*(2), 88-94. doi: 10.1016/j.advengsoft.2007.01.021.

[5]. Foukalas, F., Ntarladimas, Y., Glentis, A., & Boufidis, Z. (2005, June). Protocol reconfiguration using component-based design. In *IFIP International Conference on Distributed Applications and Interoperable Systems* (pp. 148-156). Springer, Berlin, Heidelberg. doi: 10.1007/11498094_14.

[6]. Nandaa, A. (2018). *Beginning API Development with Node. js: Build highly scalable, developer-friendly APIs for the modern web with JavaScript and Node. js*. Packt Publishing Ltd.

[7]. Andrew Mead, Learning Node.js Development, Birmingham, UK, 2018, 978-1-78839-554-0.

[8]. Raihan Taher, Hands-On Object-Oriented Programming with C#, Birmingham, UK, 2019, 978-1-78829-622-9

[9]. HKSAR Audit Commission. (2013). Office of the Government Chief Information Officer. *Audit Report Chapter*, *8*, 28.